Sistemas Informáticos
Curso 2008 - 2009

Implementación de un depurador declarativo para el lenguaje de bases de datos SQL

Nazareth Jiménez Vela

Paula Martin-Ampudia Ugena

David Sierro Elvira

Dirigido por:

Rafael Caballero Roldán

Dpto.: Sistemas Informáticos y Computación

Facultad de Informática

Universidad Complutense de Madrid

ÍNDICE

ÍNDICE DE FIGURAS Y TABLAS

ACRÓNIMOS

A

API	Application Programming Interface

B

BBDD	Bases de Datos

G

GPL	General Public License
GNU	Gnu No es Unix

J

JDBC	Java Data Connectivity

P

PHP	Hypertext Pre-processor

S

SQL	Structured Query Language
SGBD	Sistemas de Gestión de Base de Datos

AUTORIZACIÓN

Los ponentes Nazareth Jiménez Vela, con DNI 48974258J, Paula Martín-Ampudia Ugena, con DNI 50756480B, y David Sierro Elvira, con DNI 14307952C, autorizan a la Universidad Complutense de Madrid a difundir y utilizar con fines académicos tanto la propia memoria, como el código, la documentación y el prototipo desarrollado.

Nazareth Jiménez Vela

Paula Martín-Ampudia Ugena

David Sierro Elvira

En Madrid, a 3 de Julio de 2009.

RESUMEN DEL PROYECTO Y PALABRAS CLAVES

Resumen del proyecto

Los depuradores han supuesto un gran adelanto para la rama de la informática dedicada al software. Su principal objetivo consiste en depurar o limpiar los errores de otro programa informático, sirviendo esto de gran ayuda al programador. Para ello se han ido desarrollando diferentes programas depurativos dependiendo del lenguaje de programación que se quiera depurar y basados en diferentes algoritmos, intentando así conseguir, de la manera más eficiente, la limpieza de errores.

Nuestro proyecto consiste en la construcción de un Depurador Declarativo para el lenguaje declarativo SQL. Realiza la conexión con el SGBD y necesita un archivo introducido por el usuario con sentencias SQL, el cual debe contener la vista errónea principal, que será la que queremos depurar. De ella colgarán las tablas y vistas de las que dependerá. Con estos datos se genera un árbol de cómputo en el que cada nodo representa una tabla o vista de la que depende la principal. Dicho árbol será el que se recorrerá en busca de la tabla/vista errónea.

Además, se ha implementado un sistema de localización de errores, que interactuando con el usuario clasifica los nodos como válidos o inválidos y consigue determinar cuál es el nodo crítico, mostrando al usuario dónde está el error. Si el nodo crítico es una vista se realiza una búsqueda fina, mostrando al usuario el fragmento de código erróneo asociado a la vista, ayudando así a encontrar el error de la manera más aproximada posible.

Abstract

Debuggers have become a great advance for the branch of the computer science dedicated to software. Their main objective is to debug or clean errors of other computer programs, being of great help to programmers. There are different kinds of debuggers depending on the programming language chosen to debug and based in different algorithms. The purpose of this tool is to find out in an effective way the error occurring in a program.

Our project involves the creation of a declarative debugger for the declarative language SQL. It connects to the SGBD and needs a file given by the user containing the SQL sentences, which must include the wrong main view that we intend do debug. With this information a computation tree will be created. Each node in this tree represents a table or view on which the main one depends. This tree will be used to find the wrong table/view.

Furthermore, it has been implemented an error system finder that classifies with the user the nodes as valid or non valid and determines which is the buggy node, showing the user where the error is. If the buggy node is a view, a detailed search is done, showing the user the wrong fragment code associated with the view, helping to find the error in the most exact way possible.

Palabras clave

Depurador declarativo, Programación declarativa, SQL, Java, jsqlParser, top-down, divide&query, árbol de cómputo, vista, tabla, JTree, JDBC.

Capítulo 1

INTRODUCCIÓN

Comenzamos exponiendo cual ha sido la motivación que nos ha llevado a realizar nuestro proyecto de fin de carrera, una vez planteado el problema formularemos los objetivos que deseamos alcanzar, mostrando, finalmente, una visión general del resultado obtenido, es decir, nuestra solución. Además, en la última sección de este apartado describiremos, brevemente, el resto de capítulos que componen nuestra memoria.

1.1 Motivación

SQL (Structured Query Language) es un lenguaje declarativo de acceso a bases de datos relacionales que permite especificar diversos tipos de operaciones en éstas. En la actualidad SQL es el estándar *de facto* de la inmensa mayoría de los SGBD comerciales debido a las buenas características que ofrece como que es fácil de entender y de aprender, sin embargo la falta de herramientas auxiliares tales como depuradores o entornos gráficos ha sido señalada como una posible limitación. Este mismo razonamiento es aplicable en general a todos los llamados *lenguajes declarativos.*

La carencia de estas herramientas en un lenguaje no es perceptible cuando se elaboran pequeños programas experimentales sino en desarrollos complejos reales. De hecho, en el caso en el que el uso del sistema se va a limitar a programas de pocas líneas es preferible que éste sea lo más simple posible, sin embargo, en problemas reales, el uso de herramientas de apoyo hace mucho mas simple su desarrollo.

Muchas de las herramientas o técnicas empleadas habitualmente en los lenguajes imperativos no resultan aplicables en los lenguajes declarativos, por lo que adoptar estas técnicas a menudo exige el desarrollo de ideas y conceptos nuevos, como sucede en el caso de la depuración.

De aquí surge la idea de desarrollar una herramienta para la depuración declarativa de SQL.

Para profundizar en estos puntos exponemos a continuación algunas ideas claves sobre la depuración declarativa y la programación declarativa.

1.2 Depuración declarativa

Cuando se ejecuta un programa, con un objetivo dado, se esperan obtener las soluciones esperadas por el programador, quien tiene en mente el significado del programa, es decir, lo que éste debe calcular. La depuración es un proceso mediante el cual los programas detectan y señalan las causas de las discrepancias entre lo que el programa calcula realmente y lo que desea

el programador. La búsqueda de estos errores se puede enfocar desde el punto de vista del conocimiento procedural, que tiene en cuenta la secuencia de operaciones aplicadas durante la ejecución, o desde el conocimiento declarativo del programa, el cual determina cuales de los resultados devueltos por los programas son correctos.

SQL es un lenguaje declarativo, es decir, que especifica qué es lo que se quiere y no cómo conseguirlo, por lo que una sentencia no establece explícitamente un orden de ejecución; de ahí la idea de nuestro proyecto de hacer un depurador para este lenguaje.

Si consideramos detenidamente el desarrollo y construcción de depuradores para el paradigma declarativo, observaremos en seguida que la propia naturaleza de los lenguajes declarativos hace que las técnicas empleadas habitualmente, como por ejemplo que en los depuradores convencionales se permita al usuario avanzar paso a paso la evolución de un cómputo, detenerse en puntos dados y examinar el contenido de las variables, resulten difícilmente aplicables para el caso de SQL u otros lenguajes declarativos; esto se debe a que tienen un mecanismo de cómputo tan complejo que es una tarea muy compleja reproducir los detalles operaciones en la traza empleada a depurar.

Para resolver esta dificultad se considera la depuración declarativa, que se encuentra entre las diferentes alternativas encaminadas a la construcción de depuradores para lenguajes declarativos.

Su principal ventaja radica en el hecho de que sus ideas pueden abstraerse para dar lugar a un esquema general de depuración declarativa que resulta ser aplicable a todo tipo de paradigmas, incluyendo paradigmas no declarativos.

La idea fue presentada primero en el contexto de la programación lógica [7, 1, 2] y adaptada después a la programación funcional [5, 4, 6] y a la programación lógica con restricciones [8], aunque como indica Lee Naish en [3] la técnica puede ser aplicada a cualquier paradigma.

La depuración declarativa parte de la representación del cómputo a depurar mediante un árbol de cómputo adecuado. Todo nodo representará un paso de cómputo y deberá tener asociado el fragmento de código responsable de dicho paso. En particular, la raíz del árbol representará el resultado del computo principal, y el resultado asociado a cada nodo estará determinado por los resultados de sus nodos hijos, es decir, que los nodos hijos se corresponderán con los cómputos auxiliares que han sido necesarios para llegar al resultado almacenado en el nodo padre, teniendo cada uno de ellos a su vez su resultado y fragmento de código asociados.

Así, el propósito del depurador será detectar fragmentos erróneos de programa a partir del árbol de computo; para ello realizará un recorrido por el árbol con ayuda de un oráculo externo, normalmente el usuario (quien tiene un conocimiento declarativo subyacente de la semántica esperada del programa) al que se le van realizando preguntas acerca de la corrección de los nodos del árbol. Si en algún momento localiza un nodo con un resultado asociado que no es correcto, pero tal que los resultados de sus hijos si lo son, tendremos que el fragmento de código asociado a dicho nodo ha producido un resultado incorrecto a partir de datos de entrada correctos y el depurador señalará dicho fragmento indicando que es la causa del error.

A este tipo de nodos se les denomina *nodos críticos.* El depurador irá explorando el árbol de cómputo en busca solo de nodos críticos y esto es porque hay que tener en cuenta que los nodos pueden ser erróneos aun cuando su fragmento de código asociado sea correcto, ya que su resultado puede haberse obtenido partiendo de resultados ya erróneos, por eso el depurador solo detectará error cuando encuentre dichos nodos.

Diferentes tipos de errores requerirán diferentes tipos de árbol. En programación lógica se consideran dos tipos de errores susceptibles de ser tratados mediante depuración declarativa:

- Respuestas perdidas: En el conjunto de todas las respuestas obtenidas para un objetivo dado falta alguna respuesta esperada. Un caso particular de esta situación es cuando no se obtiene ninguna respuesta cuando se esperaba al menos una.

- Respuestas incorrectas: Se obtiene una respuesta inesperada para un objetivo determinado.

1.2.1 Programación declarativa

La **Programación Declarativa**, es un paradigma de programación que está basado en el desarrollo de programas especificando o "declarando" un conjunto de condiciones, proposiciones, afirmaciones, restricciones, ecuaciones o transformaciones que describen el problema y detallan su solución. La solución es obtenida mediante mecanismos internos de control, sin especificar exactamente cómo encontrarla.

En la programación declarativa las sentencias que se utilizan lo que hacen es **describir el problema** que se quiere solucionar, pero no las instrucciones necesarias para solucionarlo. Esto último se realizará mediante mecanismos internos de inferencia de información a partir de la descripción realizada.

Una de las ventajas de este enfoque es que el programador solo debe describir cuales son las características del problema, sin tener que preocuparse en detalle de como los cómputos son llevados a la practica. Además, el alto nivel de abstracción de estos lenguajes permite probar formalmente la corrección de los programas y razonar acerca sus propiedades de forma sencilla. Estas ideas han dado lugar a dos corrientes distintas: la programación lógica y la programación funcional.

1.2.2 Otros paradigmas declarativos

La **Programación lógica** se basa en fragmentos de la lógica de predicados, representa sus programas como formulas lógicas, en particular como cláusulas, y emplea mecanismos también

propios de la lógica matemática para la resolución de objetivos. El lenguaje de programación lógica por excelencia es Prolog, que cuenta con diversas variantes. La más importante es la *programación lógica con restricciones,* que posibilita la resolución de ecuaciones lineales además de la demostración de hipótesis.

Los lenguajes de **Programación funcional e**stán enraizados en el concepto de función (matemática) y su definición mediante ecuaciones (generalmente recursivas), que constituyen el programa. Entre los lenguajes de este tipo mas conocidos se encuentran Haskell.

También cabe destacar la existencia de la **Programación lógico-funcional,** un solo paradigma que combina las principales ventajas de la programación lógica y de la programación funcional.

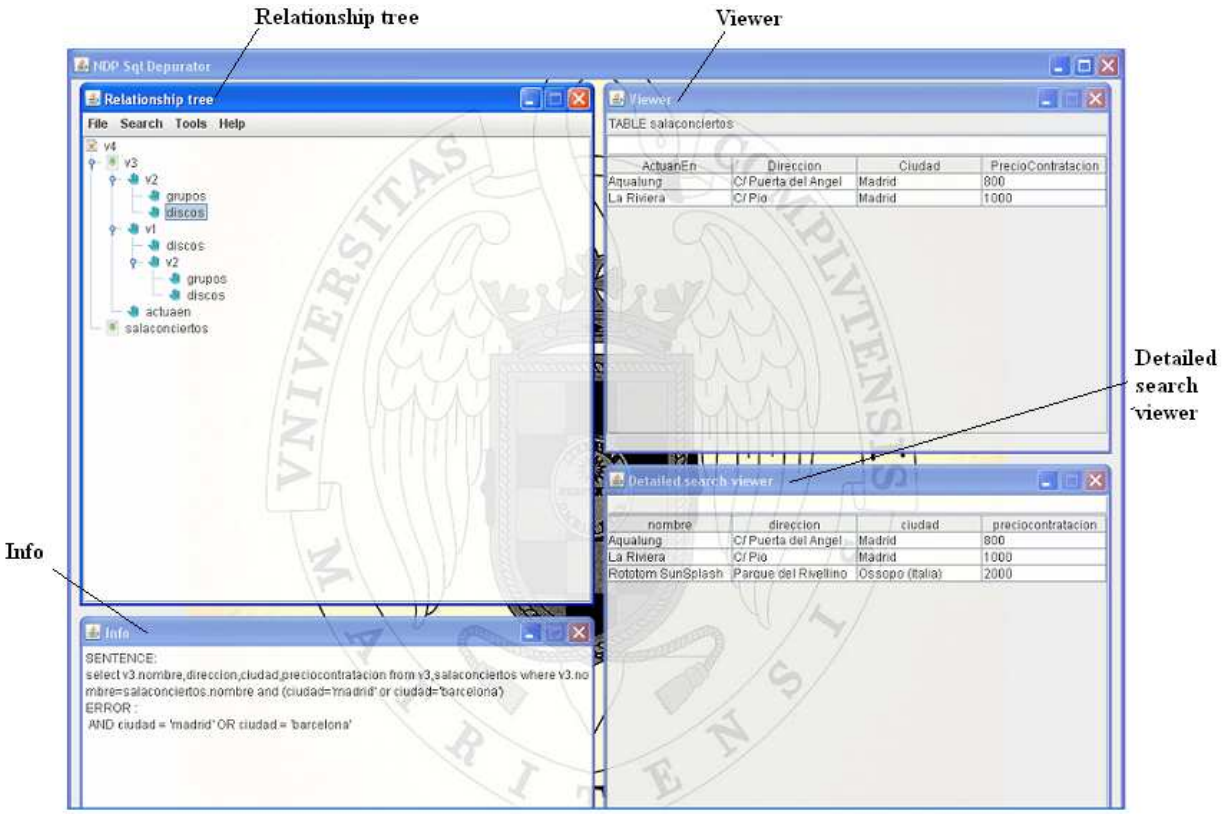
1.3 Objetivos

El objetivo de nuestro proyecto es implementar un depurador declarativo para programas SQL.

La ejecución del programa crea a partir de un archivo SQL un árbol de dependencias de evaluación como árbol de cómputo, una estructura, como hemos visto, basada en la idea de la depuración declarativa. Una sesión de depuración consistirá en una serie de preguntas y respuestas. Las preguntas son propuestas por el depurador, y las respuestas provistas por el usuario. Como consecuencia de dicha sesión nuestro depurador encontrará el error que estará asociado bien a una tabla o bien a una vista. En el caso de que esté asociado a una vista, el error en sí estará en el código de dicha vista, mientras que si el error se señala en una tabla, éste estará en el conjunto de los datos almacenados por dicha tabla (es decir, estos datos no serán los esperados por el usuario). Por estos motivos si el error se encuentra en una tabla la solución será directa mientras que si el error se da en una vista podremos empezar a realizar una búsqueda detallada en el código asociado.

A continuación mostramos lo que sería el aspecto de nuestra herramienta en ejecución, una vez cargado el archivo SQL que se quiere depurar.

Figura 1. Herramienta en ejecución.



Como se puede observar la herramienta se compone de 4 ventanas:

- **Relationship tree:** Muestra el árbol de dependencias creado a partir del archivo SQL. Nos servirá de guía para depurar con los distintos métodos de búsqueda de error.
- **Viewer:** Muestra el contenido de la tabla o vista del nodo seleccionado en la ventana *Relationship tree*.
- **Detailed search viewer:** Una vez el depurador haya encontrado un nodo crítico, la aplicación ofrece una búsqueda detallada, basada en preguntas al usuario, para así poder hallar el error más aproximado en el fragmento de código asociado a dicho nodo. Inicialmente esta ventana muestra el resultado de dicho código que corresponde al contenido de la vista asociada al nodo crítico, que se irá modificando a medida que el usuario vaya respondiendo a las preguntas propuestas por el depurador.
- **Info:** Como hemos explicado, durante la depuración se realizan una serie de preguntas las cuales se muestran en este panel; acabada la depuración y encontrado el fallo en esta

ventana aparecerá el fragmento de código asociado al nodo crítico y el error exacto en dicho fragmento.

1.4 Estructura de la memoria

El siguiente capitulo detallará las características de los lenguajes escogidos para nuestro proyecto y el por qué de su elección. Además se realizará una descripción detallada de los casos de uso de nuestra aplicación así como de los requisitos funcionales y no funcionales para la correcta elaboración de los mismos. Para finalizar el capitulo se utilizará el lenguaje de modelado UML para definir nuestro sistema describiendo algunos de los procesos realizados en él.

En el capitulo 3 se describirá la implementación realiza para llevar a cabo nuestro sistema. En la primera parte de dicho capitulo se realizará un ejemplo de ejecución de la herramienta mientras se explica internamente como la hemos programado. A continuación se hablará de los problemas que nos han surgido a lo largo de toda la implementación, de las limitaciones de nuestro sistema y de las posibles extensiones que pueden tener lugar en un trabajo posterior sobre nuestro sistema.

Para finalizar, el capitulo 4 se centrará en los distintos métodos de navegación de los árboles de depuración, comentando sus ventajas y desventajas así como las diferentes estrategias de navegación.

Capítulo 2

ANÁLISIS Y DISEÑO

Nuestro objetivo ha sido realizar un depurador de un sistema de gestión de base de datos, el escogido ha sido MySQL, sistema de gestión de bases de datos relacional, licenciado bajo la GPL de la GNU, y el lenguaje de programación utilizado ha sido Java, orientado a objetos y licenciado también bajo la GPL de la GNU.

2.1 BBDD: MySQL.

El motivo que nos ha impulsado a escoger MySQL es principalmente que está desarrollado bajo la filosofía de código abierto y que es multiplataforma, aunque posee grandes características.

Su diseño multihilo le permite soportar una gran carga de forma muy eficiente. Es el sistema de gestión más rápido, los desarrolladores sostienen que MySQL es posiblemente la base de datos más rápida que pueden encontrar. En cuanto a facilidad de uso es un sistema de alto rendimiento pero relativamente simple y es mucho menos complejo de configurar y administrar que sistemas más grandes, además comprende SQL (structured query lenguaje, lenguaje de consulta estructurado), que es el lenguaje utilizado para todos los sistemas de bases de datos modernos.

Con respecto a capacidad pueden conectarse muchos clientes simultáneamente al servidor. Los clientes pueden utilizar varias bases de datos simultáneamente. Pueden acceder de forma interactiva a MySQL empleando diferentes interfaces que le permitan introducir consultas y visualizar los resultados: cliente de línea de comando, navegadores Web o clientes del sistema por Windows. Además, está disponible una amplia variedad de interfaces de programación para lenguajes como C, Perl, Java, PHP Y PITÓN. Por tanto, tiene la posibilidad de elegir entre usar un software cliente-preempaquetado o escribir sus propias aplicaciones a medida.

Por otro lado, MySQL está completamente preparado para el trabajo en red y las bases de datos pueden ser accedidas desde cualquier lugar en Internet, por lo que puede compartir sus datos con cualquiera, en cualquier parte, pero MySQL dispone de control de acceso, un sistema de privilegios y contraseñas que es muy flexible y seguro, y que permite verificación basada en el Host. Las contraseñas son seguras porque todo el tráfico de contraseñas está encriptado cuando se conecta con un servidor. Tiene gran portabilidad, se ejecuta en muchas variantes de UNIX, así como en otros sistemas no-UNIX, como Windows y OS/2. MySQL se ejecuta en

hardware que va desde PC hasta servidores de alta capacidad. Además de todo esto, es de distribución abierta y fácil de conseguir.

Aunque finalmente hemos preferido usar MySQL, principalmente por ser software libre lo suficientemente potente, nuestro programa sería fácil de modificar para trabajar con otro SGBD, como Oracle, Access, etc.

2.2 Lenguaje de programación: Java.

Las características principales que nos ofrece el lenguaje de programación utilizado, Java, respecto a cualquier otro, serían:

- **Simple:** No es necesario preocuparse de liberar memoria, el reciclador se encarga de ello y como es de baja prioridad, cuando entra en acción, permite liberar bloques de memoria muy grandes, lo que limita en mucho la fragmentación de la memoria.

- **Orientado a objetos:** Java trabaja con sus datos como objetos y con interfaces a esos objetos. Soporta las tres características propias del paradigma de la orientación a objetos: encapsulación, herencia y polimorfismo.

- **Distribuido:** Java en sí no es distribuido, sino que proporciona las librerías y herramientas para que los programas puedan ser distribuidos, es decir, que se corran en varias máquinas, interactuando.

- **Robusto:** Java realiza verificaciones en busca de problemas tanto en tiempo de compilación como en tiempo de ejecución. La comprobación de tipos en Java ayuda a detectar errores, lo antes posible, en el ciclo de desarrollo. Java obliga a la declaración explícita de métodos, reduciendo así las posibilidades de error. Maneja la memoria para eliminar las preocupaciones por parte del programador de la liberación o corrupción de memoria.

  También implementa los *arrays* auténticos, en vez de listas enlazadas de punteros, con comprobación de límites, para evitar la posibilidad de sobreescribir o corromper memoria resultado de punteros que señalan a zonas equivocadas. Estas características reducen drásticamente el tiempo empleado en el desarrollo de aplicaciones Java.

  Además, para asegurar el funcionamiento de la aplicación, realiza una verificación de los *ByteCodes*, que son el resultado de la compilación de un programa Java. Es un código de máquina virtual que es interpretado por el intérprete Java. No es el código máquina directamente entendible por el hardware, pero ya ha pasado todas las fases del compilador: análisis de instrucciones, orden de operadores, etc., y ya tiene generada la pila de ejecución de órdenes.

  Java proporciona, pues:

  - Comprobación de punteros
  - Comprobación de límites de arrays
  - Excepciones
  - Verificación de ByteCodes

- **Arquitectura neutral – Independencia de la plataforma:** Para establecer Java como parte integral de la red, el compilador Java compila su código a un fichero objeto de formato independiente de la arquitectura de la máquina en que se ejecutará. Cualquier máquina que tenga el sistema de ejecución (*run-time*) puede ejecutar ese código objeto, sin importar en modo alguno la máquina en que ha sido generado.

- **Seguro:** Los niveles de seguridad que presenta son:

  - Fuertes restricciones al acceso a memoria, como son la eliminación de punteros aritméticos y de operadores ilegales de transmisión.
  - Rutina de verificación de los *códigos de byte* que asegura que no se viole ninguna construcción del lenguaje.
  - Verificación del nombre de clase y de restricciones de acceso durante la carga.
  - Sistema de seguridad de la interfaz que refuerza las medidas de seguridad en muchos niveles.

- **Portable:** Más allá de la portabilidad básica por ser de arquitectura independiente, Java implementa otros estándares de portabilidad para facilitar el desarrollo. Los enteros son siempre enteros y además, enteros de 32 bits en complemento a 2. Además, Java construye sus interfaces de usuario a través de un sistema abstracto de ventanas de forma que las ventanas puedan ser implantadas en entornos Unix, PC o Mac.

- **Interpretado:** El intérprete Java (sistema *run-time*) puede ejecutar directamente el código objeto.

- **Multihilo:** Al ser MultiHilo, Java permite muchas actividades simultáneas en un programa. El beneficio de ser multihilo consiste en un mejor rendimiento interactivo y mejor comportamiento en tiempo real.

- **Dinámico:** Java no intenta conectar todos los módulos que comprenden una aplicación hasta el mismo tiempo de ejecución. Las librerías nuevas o actualizadas no paralizarán la ejecución de las aplicaciones actuales, siempre que mantengan el API anterior.

Java también simplifica el uso de protocolos nuevos o actualizados. Si su sistema ejecuta una aplicación Java sobre la red y encuentra una pieza de la aplicación que no sabe manejar, tal como se ha explicado en párrafos anteriores, Java es capaz de traer automáticamente cualquier pieza que el sistema necesite para funcionar.

Además de todas estas ventajas cabe destacar la forma estándar de conectar con bases de datos que proporciona este lenguaje de programación, algo que nos hacía más fácil la conexión con MySQL. También fue bastante influyente la librería parsing ya hecha en lenguaje Java.

2.3 Casos de uso.

A continuación, vamos a realizar una descripción detallada de los casos de uso así como de los requisitos funcionales y no funcionales para la correcta elaboración de los mismos, dirigidos tanto a los desarrolladores como a las personas encargadas de testear el producto.

2.3.1 Resumen de Casos de Uso

- Ayuda: Añadir/modificar habitaciones y modificar tarifas del hotel.
- Búsqueda manual: Búsqueda manual de los errores en las vistas.
- Búsqueda algorítmica: Búsqueda algorítmica de los errores en las vistas.
- Desplegar&Contraer: Desplegar/Contraer árbol de vistas y tablas.
- Preferencias: Configurar preferencias.
- Cargar: Cargar datos de fichero y crear árbol de vistas.
- Mostrar BBDD y consola: Mostrar las tablas y vistas de la base de datos y ejecutar órdenes en MySQL.

2.3.2 Supuestos y dependencias

El proyecto se ha visto limitado por las funciones ofrecidas por el jsqlParser [10] ofrecido por sourceforge. Dichas limitaciones están detalladas en esta memoria en cada versión.

2.3.3 Requerimientos específicos

Tabla 1. Ayuda

CASO DE USO #Help	Ayuda
Objetivo en contexto	Ver la ayuda del sistema
Precondiciones	Haber iniciado el sistema y existencia del fichero de ayuda en su lugar por defecto
Secuencia normal	**Paso.- Acción**
	1.- Seleccionar botón Help
	2.- Se muestran la información existente en el fichero en la ruta por defecto

Tabla 2. Búsqueda Manual

CASO DE USO #BuMa	Búsqueda Manual
Objetivo en contexto	Búsqueda manual de errores en el árbol de vistas y tablas
Precondiciones	Haber cargado un fichero válido
Secuencia normal 1.	**Paso.- Acción**
	1.- Selección botón "manual" en el submenú "search"
	2.- Aparición panel de asignación de estado a nodos
	3.- Seleccionar nodo
	4.- Pulsar estado a asignar.
	5.- Comprobar si hay errores. Si no hay errores volver a 3
	6.- Error encontrado. Si es vista: Responder si se desea búsqueda detallada. Respuesta afirmativa pasar a S2
Secuencia S2.	**Paso.- Acción**
	Búsqueda detallada
	1.- Responder pregunta detallada sobre vista
	2.- Comprobar posibles soluciones. Si no encontradas volver a 1
	3.- Solución encontrada: Mostrar solución

Tabla 3. **Búsqueda algorítmica**

CASO DE USO #BuAl	Búsqueda algorítmica
Objetivo en contexto	Búsqueda algorítmica de errores en el árbol de vistas y tablas
Precondiciones	Haber cargado un fichero válido
Secuencia normal	**Paso.- Acción**
	1.- Selección botón "top-down" o "divide&query" en el submenú "search"
	2.- Responder cuestión sobre vista mostrada
	3.- Comprobar errores. Si no hay errores volver a 2 con otra vista
	4.- Error encontrado. Si es vista: Responder si se desea búsqueda detallada. Respuesta afirmativa pasar a S2
Secuencia S2.	**Paso.- Acción**
	Búsqueda detallada
	1.- Responder pregunta detallada sobre vista
	2.- Comprobar posibles soluciones. Si no encontradas volver a 1
	3.- Solución encontrada: Mostrar solución

Tabla 4. **Mostrar BBDD y Consola**

CASO DE USO #D&U	Mostrar BBDD y Consola
Objetivo en contexto	Mostrar las tablas y vistas de la base de datos actual y ejecutar órdenes en MySQL
Precondiciones	Estar correctamente conectado con la base de datos
Secuencia normal	**Paso.- Acción**
	Mostrar tablas/vistas
	1.- Seleccionar botón "Show Database tables"
	2.- Visualizar información en panel de tablas y vistas
Secuencia normal 2	**Paso.- Acción**
	Ejecutar orden
	1.- Ejecutar instrucción sql en consola del panel de tablas y vistas
	2.- Visualizar información en panel de tablas y vistas

Tabla 5. Desplegar & Contraer

CASO DE USO #D&U	Desplegar&Contraer
Objetivo en contexto	Desplegar o contraer el árbol de vistas
Precondiciones	Haber cargado un fichero válido
Secuencia normal	**Paso.- Acción**
	Desplegar árbol
	1.- Seleccionar botón "deploy" para desplegar el árbol
Secuencia normal 2	**Paso.- Acción**
	Contraer árbol
	1.- Selecciona botón "undeploy" para contraer el árbol

Tabla 6. Cargar

CASO DE USO #Load	Cargar
Objetivo en contexto	Cargar un archivo y crear árbol de vistas
Secuencia normal	**Paso.- Acción**
	1.- Seleccionar del submenú "File" la opción "Load"
	2.- Elegir la ruta del fichero deseado
	3.- Visualizar árbol de vistas

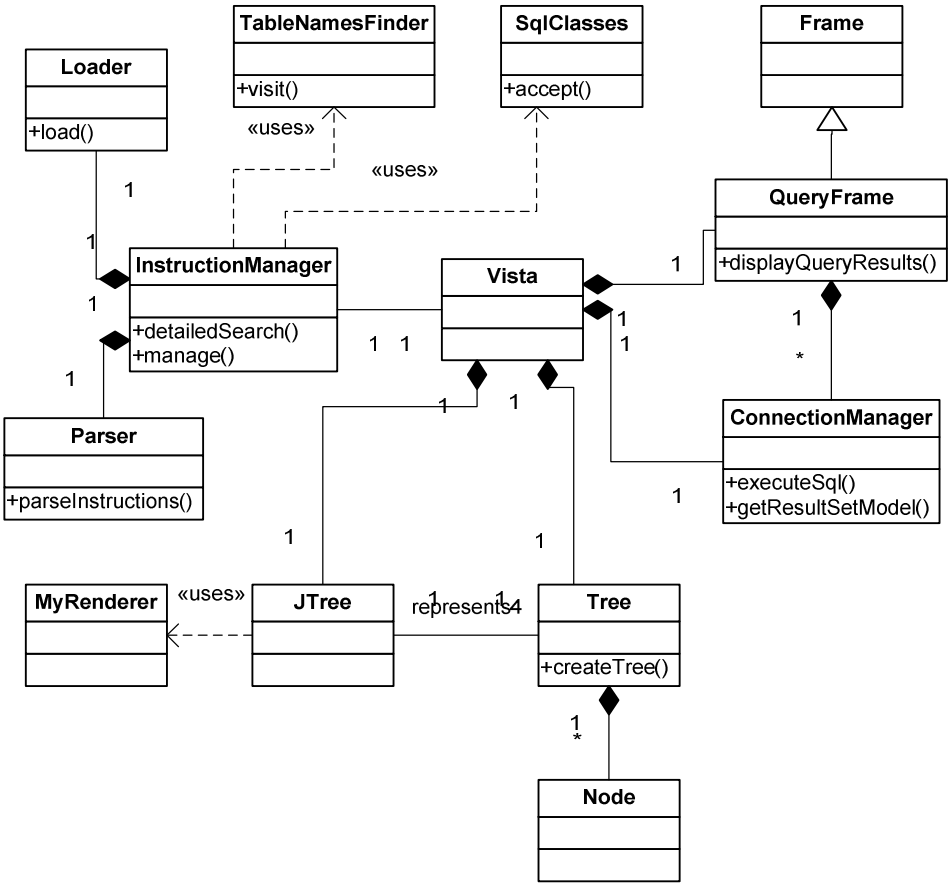
Tabla 7. Preferencias

CASO DE USO #Pref	Preferencias
Objetivo en contexto	Configurar preferencias
Secuencia normal	**Paso.- Acción**
	1.- Introducir usuario mysql
	2.- Introducir contraseña mysql
	3.- Introducir nombre de la base de datos
	4.- Elegir recordar datos o no
	5.- Pulsar "save&connect". Si éxito, finalizado, si no pasar a 6
	6.- Mostrar mensaje de error

2.4 Diagramas UML.

2.4.1 Diagramas de clases

Figura 2. Diagramas de clases.



2.4.2 Diagramas de secuencias y actividad.

Figura 3. Diagrama de actividad de Ayuda

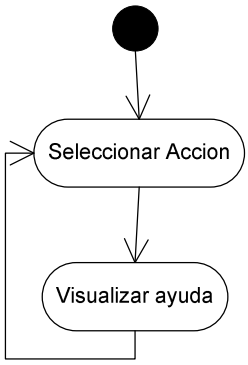


Figura 4. Diagrama de secuencias de ayuda.

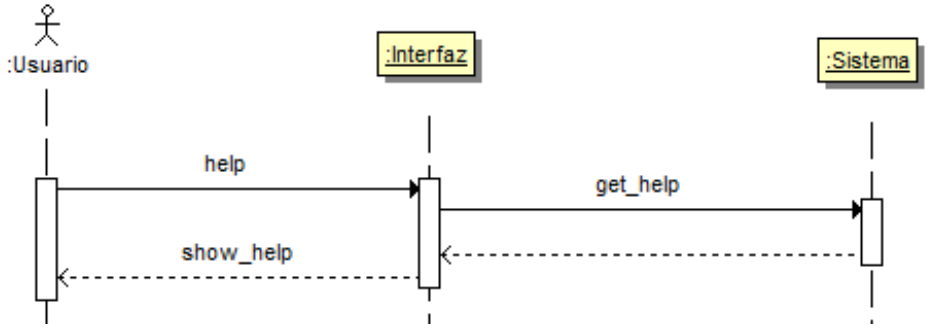


Figura 5. Diagrama de actividad de la búsqueda manual.

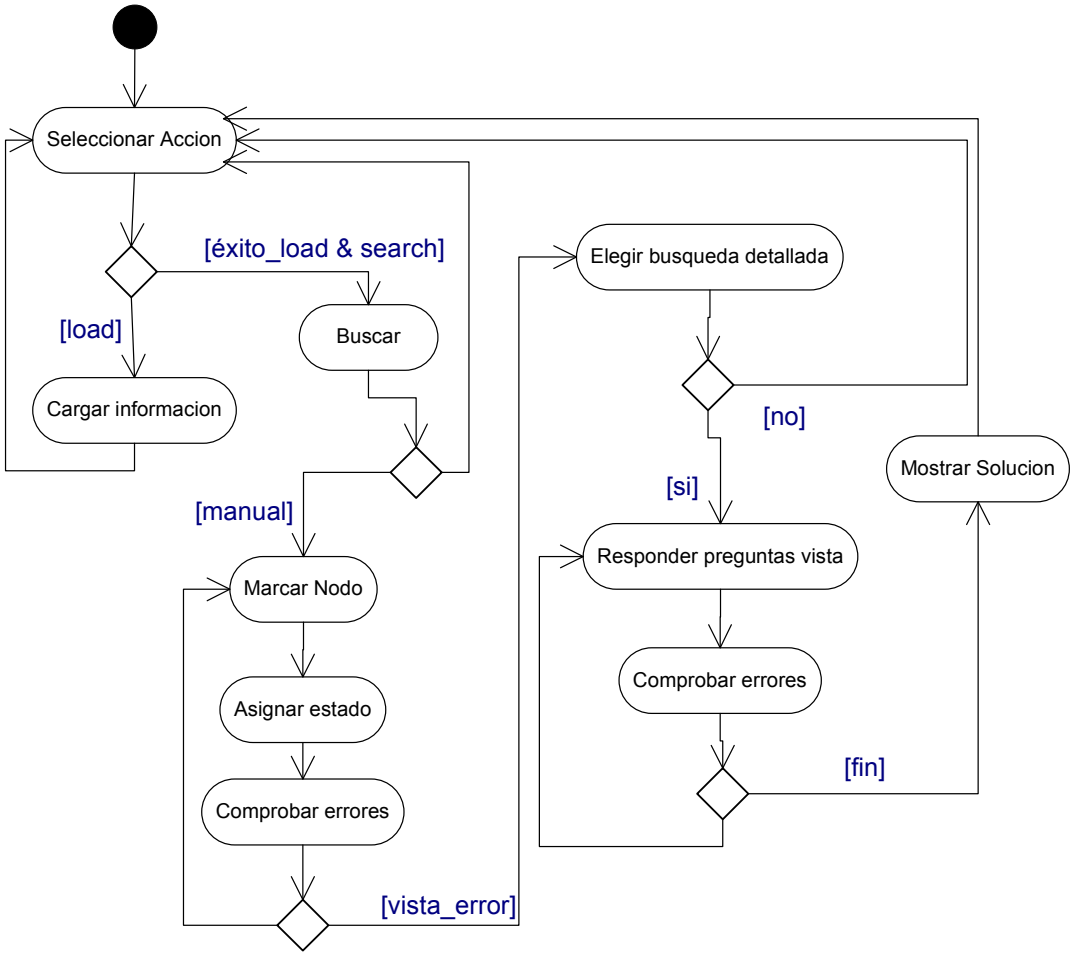


Figura 6. Diagrama de secuencias de la búsqueda manual.

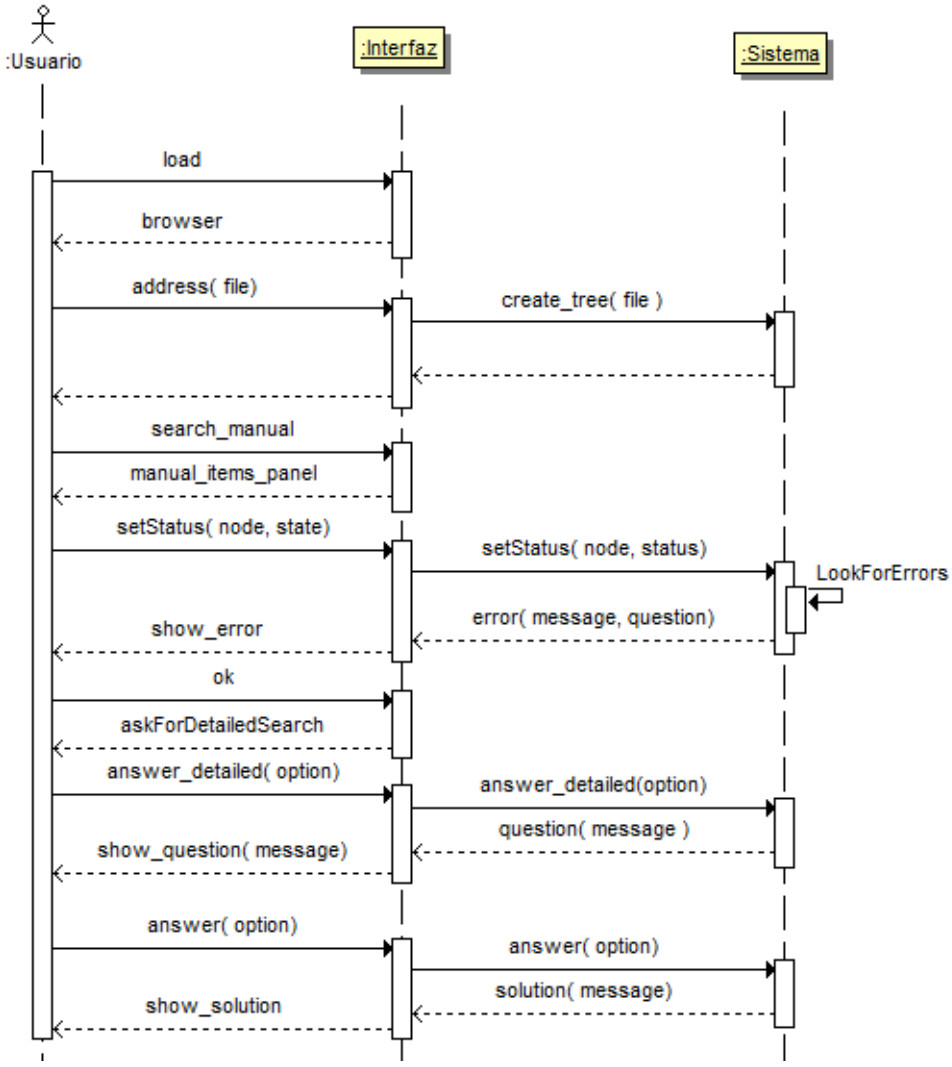


Figura 7. Diagrama de actividad de la búsqueda algorítmica.

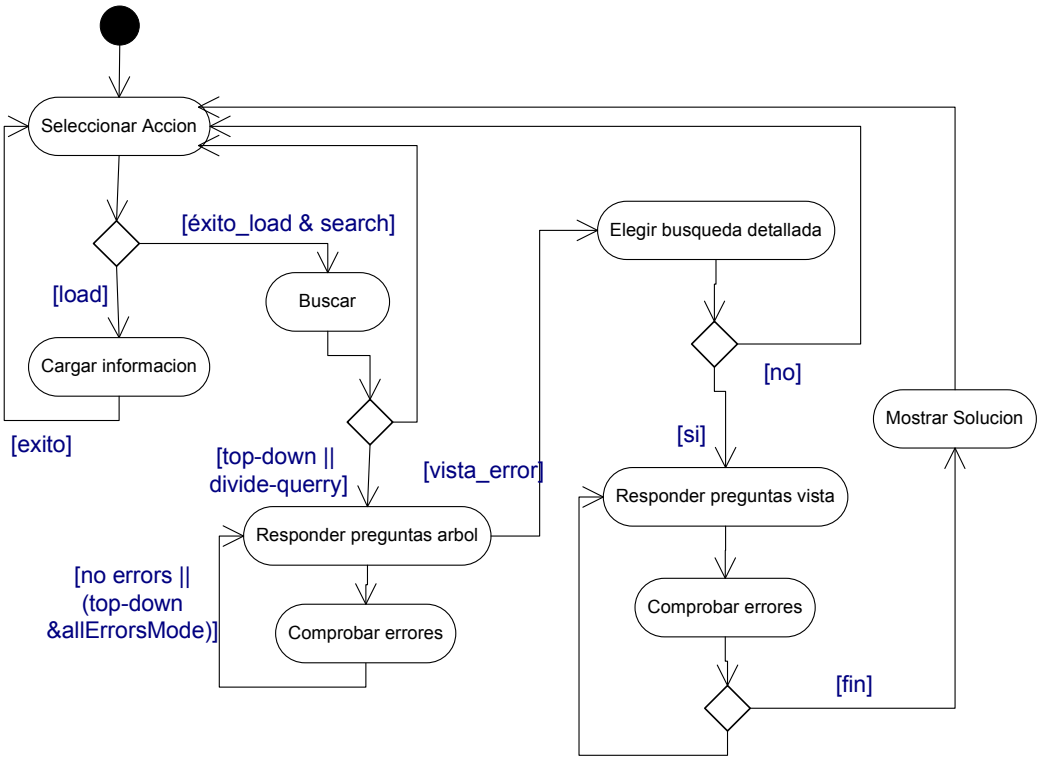


Figura 8. Diagrama de secuencias de la búsqueda algorítmica.

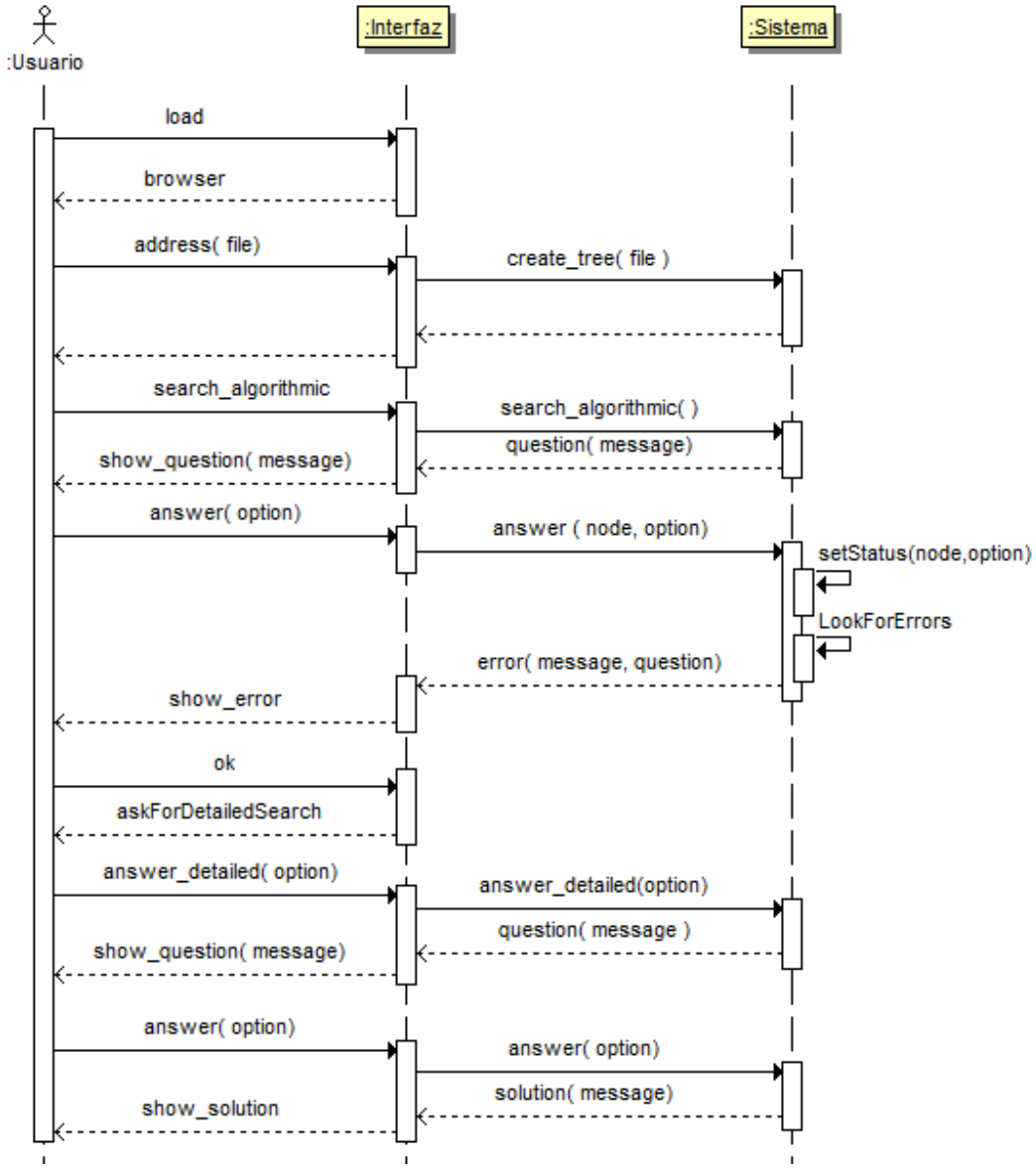


Figura 9. Diagrama de actividad de mostrar BBDD.

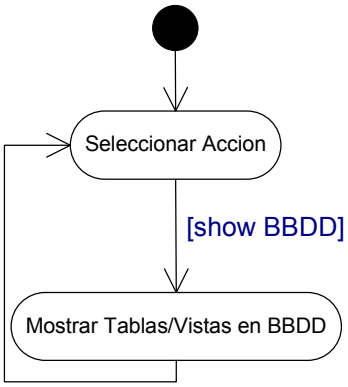


Figura 10. Diagrama de secuencias de mostrar BBDD

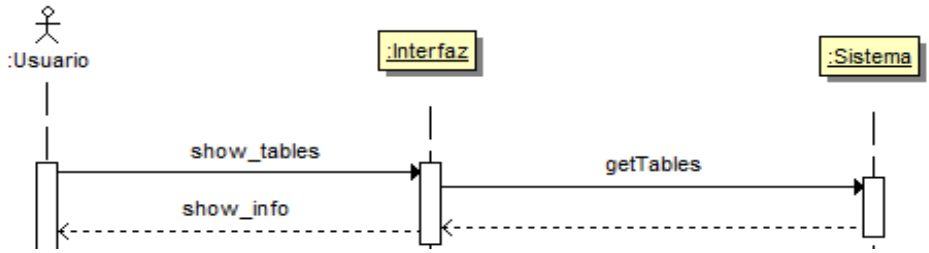


Figura 11. Diagrama de actividad de cargar.

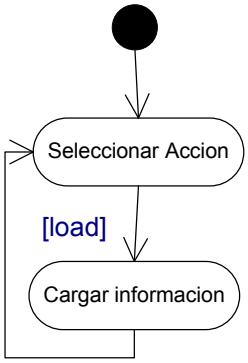


Figura 12. Diagrama de secuencias de cargar.

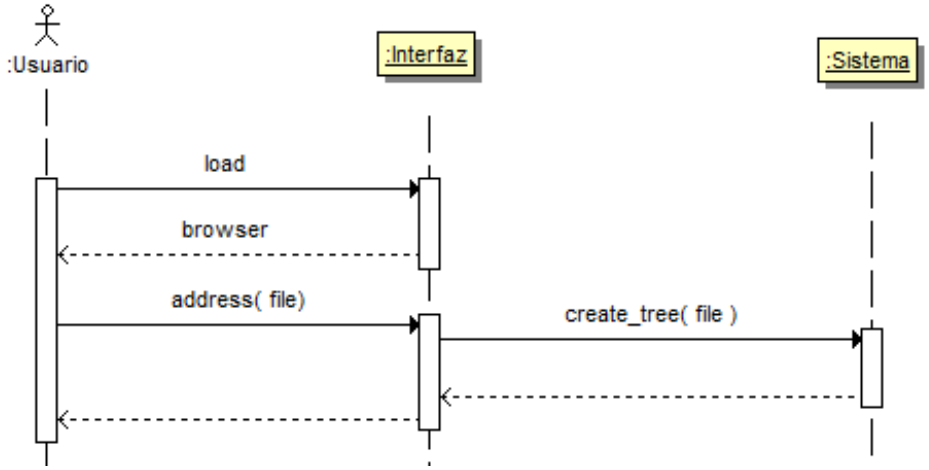


Figura 13. Diagrama de actividad de Desplegar&Contraer.

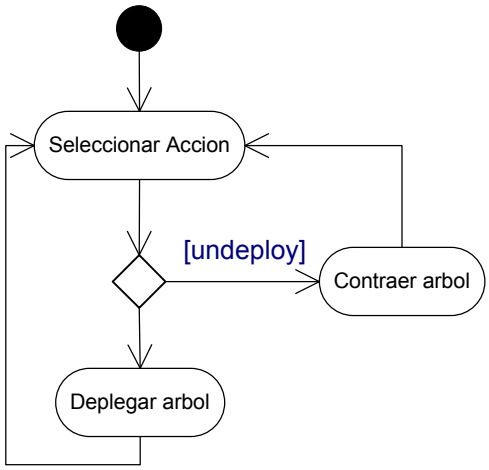


Figura 14. Diagrama de secuencias de Desplegar&Contraer.

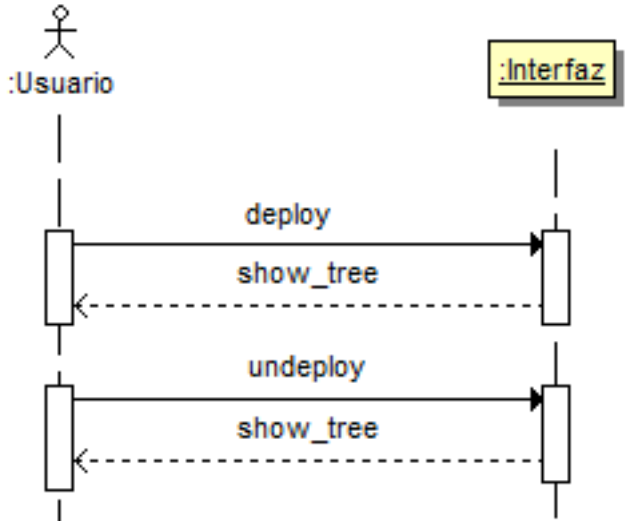


Figura 15. Diagrama de actividad de Preferencias.

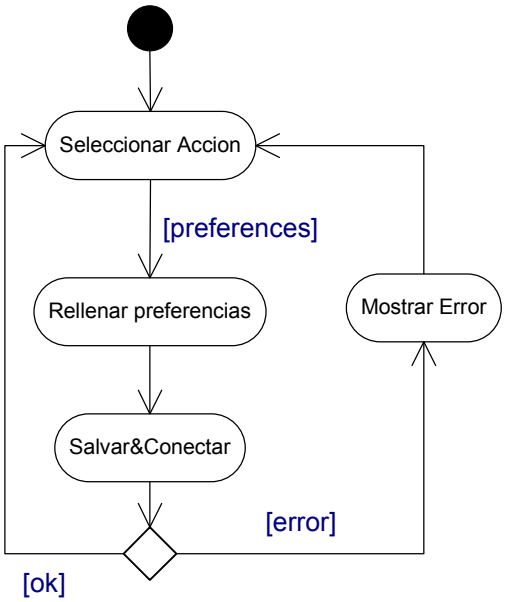
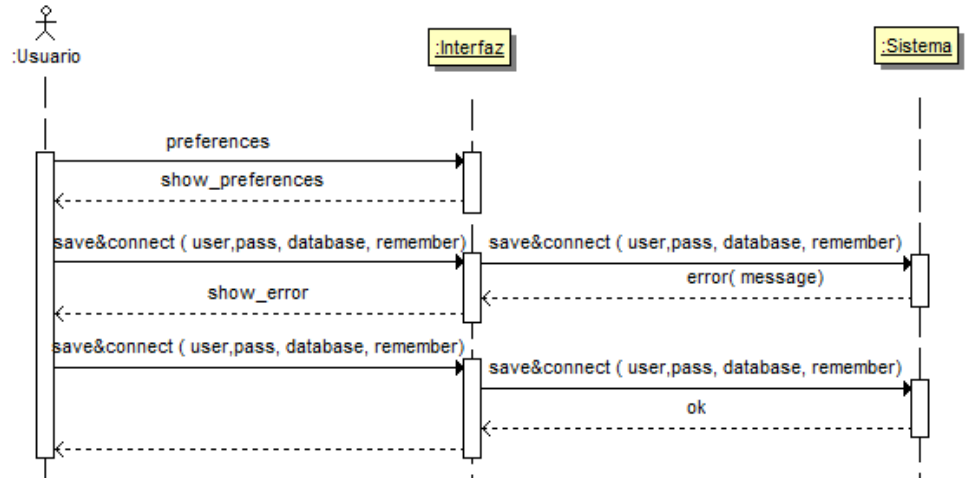


Figura 16. Diagrama de actividad de Preferencias.



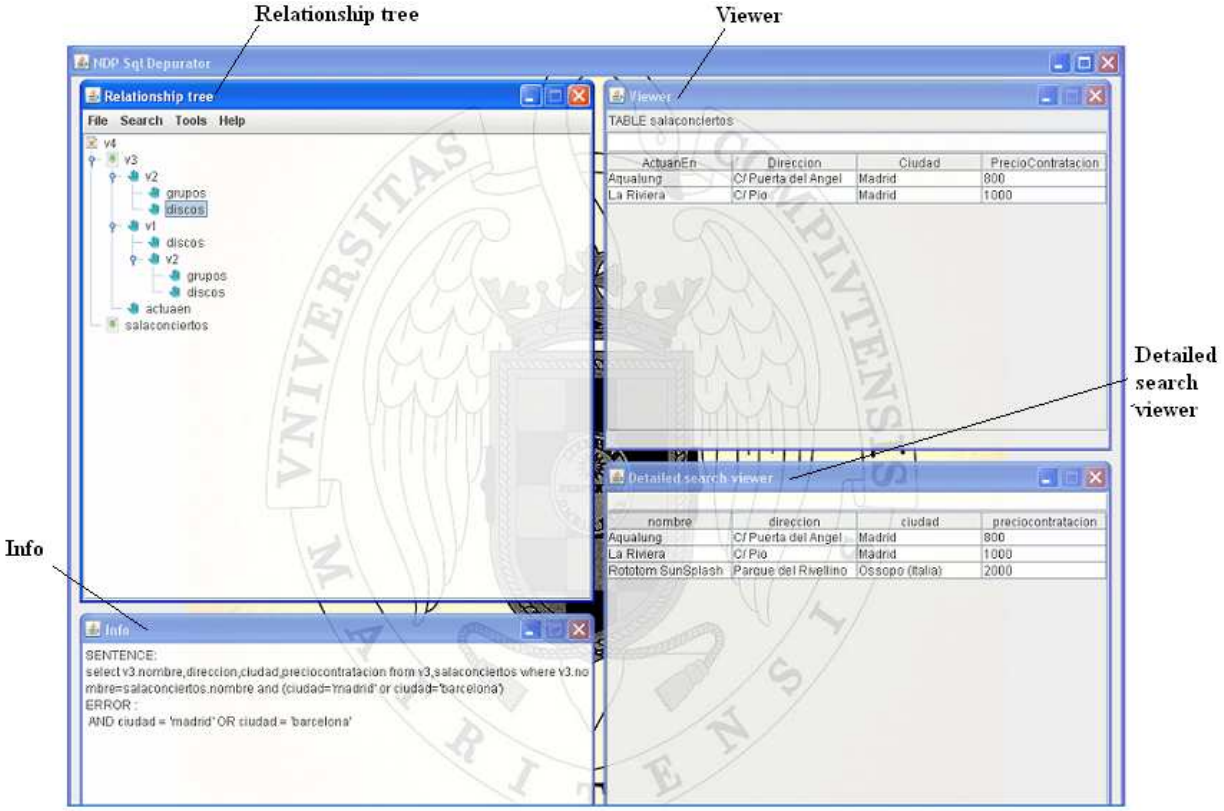
Capítulo 3

IMPLEMENTACIÓN

En este capítulo nos encargaremos de describir la arquitectura de nuestra herramienta intercalándolo a su vez con un ejemplo de ejecución de la misma, que servirá a la vez de manual. Para poder describir dicha arquitectura haremos uso de las distintas clases java que tiene nuestro proyecto, explicándolas en la medida necesaria.

Antes de nada, vamos a repasar el objetivo de cada una de las principales ventanas de nuestra aplicación:

Figura 17. Interfaz de la herramienta



Como recordaremos la herramienta se compone de 4 ventanas:

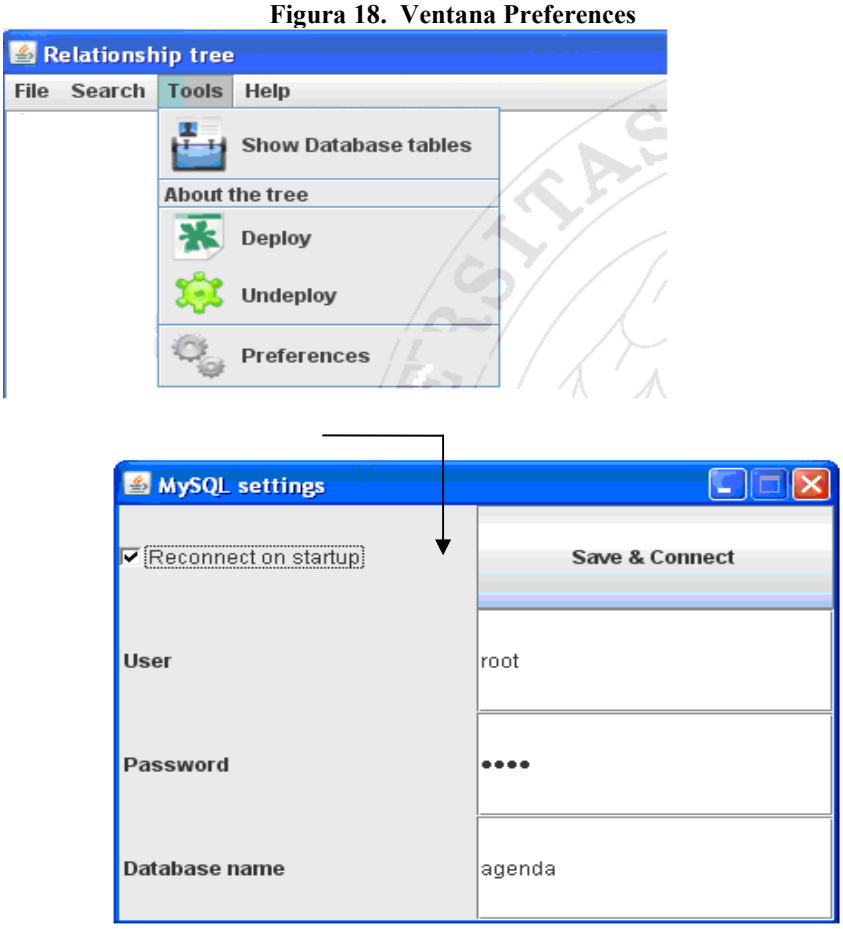
- **Relationship tree:** Muestra el árbol de dependencias creado a partir del archivo SQL. Nos servirá de guía para depurar con los distintos métodos de búsqueda de error.
- **Viewer:** Muestra el contenido de la tabla o vista del nodo seleccionado en la ventana *Relationship tree*.

- **Detailed search viewer:** Una vez el depurador haya encontrado un nodo crítico, la aplicación ofrece una búsqueda detallada, basada en preguntas al usuario, para así poder hallar el error exacto en el fragmento de código asociado a dicho nodo. Inicialmente esta ventana muestra el resultado de dicho código que corresponde al contenido de la vista asociada al nodo crítico, que se irá modificando a medida que el usuario vaya respondiendo a las preguntas propuestas por el depurador.
- **Info:** Como hemos explicado, durante la depuración se realizan una serie de preguntas las cuales se muestran en este panel; acabada la depuración y encontrado el fallo en esta ventana aparecerá el fragmento de código asociado al nodo crítico y el error exacto en dicho fragmento.

3.1 Análisis de la herramienta.

La primera tarea a realizar es la de la conexión con la base de datos que queremos utilizar. Para ello utilizamos la opción del menú **Tools→Preferences** la cual nos mostrará una ventana para que introduzcamos el usuario, la contraseña y el nombre de la base de datos correspondiente.

Figura 18. Ventana Preferences

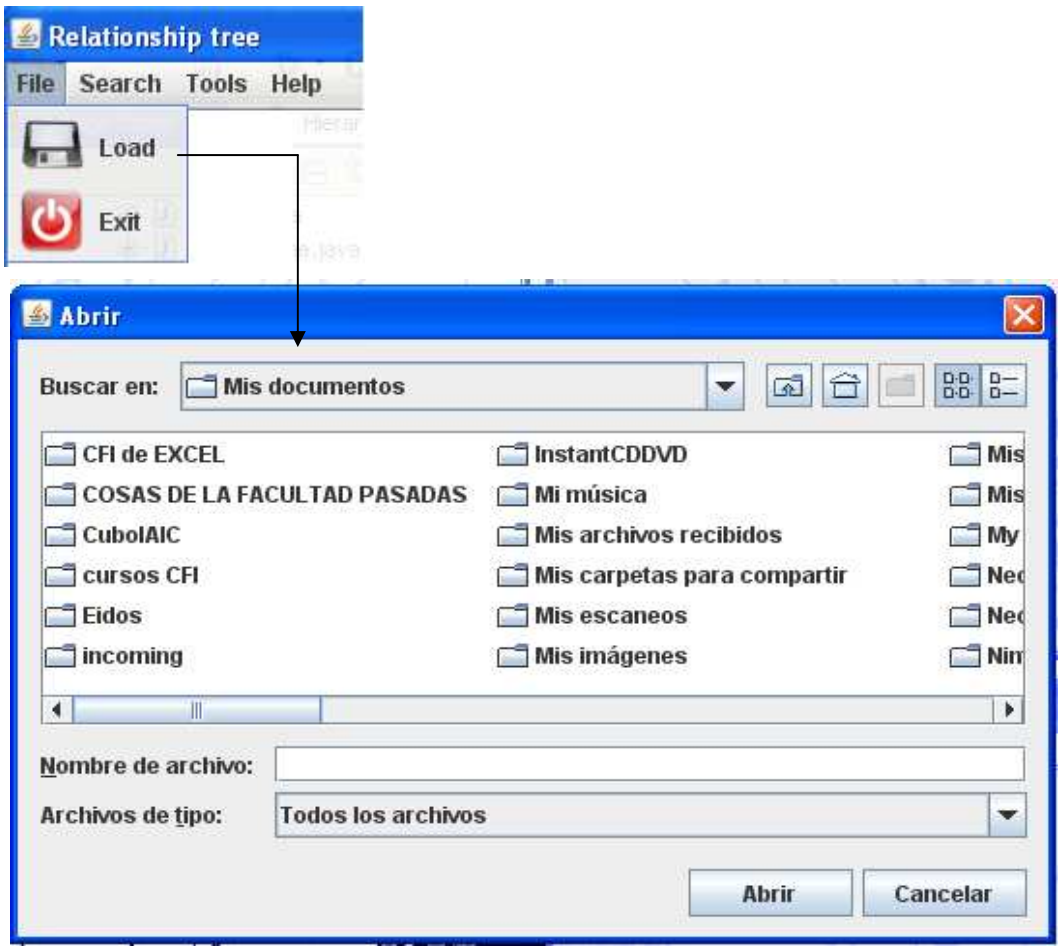


La opción "*Reconnect on startup"* evita al usuario repetir este paso, guardando todos estos datos en memoria para que la próxima vez que se entre en la herramienta la conexión se realice de forma automática.

La clase encargada de dotar de dicha funcionalidad a nuestra herramienta es *"ConnectionManager.java"*; en ella utilizamos la API *"Java Data Connectivity"* más conocida por sus siglas **JDBC**, ésta API permite la ejecución de operaciones sobre bases de datos desde el lenguaje de programación Java, independientemente del sistema operativo donde se ejecute o de la base de datos a la cual se accede. Cabe decir que esta API es utilizada a lo largo de todo nuestro proyecto para poder así no solo establecer una conexión con la base de datos deseada por el usuario, como es en el caso en esta clase, sino para también hacer todas las operaciones necesitadas en ella. Por último decir que para poder guardar los datos y acceder a ellos en el momento de realizar la conexión (tanto automática como no) esta clase hace uso de un archivo llamado *config.ini* , en el cual guardamos los datos almacenados por el usuario en MySQLSettings.

Como siguiente tarea a realizar es la de la carga del fichero que contiene las sentencias SQL que se desean depurar. Para ello utilizamos la opción del menú **File→ Load**

Figura 19. Ventana Load



Desde el punto de vista del usuario la carga de su archivo consistirá simplemente en seleccionar su archivo en la ventana arriba mostrada, sin embargo por dentro de la herramienta esta acción conlleva a muchas otras. A continuación explicamos dicho proceso.

En nuestro proyecto la acción de cargar como hemos dicho conlleva varias tareas:

- Comprobar que es un archivo válido, con válido nos referimos a que sea un archivo SQL y con una determinada estructura, la cual la explicaremos en el apartado de limitaciones ya que en este paso lo podemos obviar.
- Parsear el archivo, es decir, dividirlo en instrucciones y manejar cada una de ellas para crear el árbol de dependencias.

Las clases que implementan dichas funcionalidades son varias:

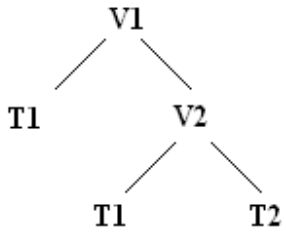
- *Loader.java*: es la clase encargada de implementar la carga del archivo, para ello lo leerá línea por línea y lo almacenará en un *string*. Este *string* se dividirá en las distintas instrucciones y se almacenará en un array de string para su posterior manejo. Esta tarea se realiza por la clase *Parser.java* que explicamos a continuación.
- *Parser.java*: esta clase tiene diferentes métodos que analizan/parsean las distintas instrucciones para así poder trabajar con ellas a posteriori.
- *Tree*.java: esta clase se encarga de crear un árbol que represente las dependencias entre las distintas instrucciones. El archivo SQL que introduzca el usuario tendrá varias vistas que pueden dependen a la vez entre sí o depender de otras tablas ya creadas en la base de datos. Las tablas en el árbol se corresponderán con nodos hoja ya que no dependerán de nadie mientras que las vistas serán nodos intermedios que a su vez tendrán nodos hijos y su respectivo nodo padre. La raíz del árbol será la vista principal del archivo SQL cargado. Así si tuviésemos en el archivo las instrucciones:

  create view v2 as (select A,B,C from T1,T2);

  create view v1 as (select A,B from T1,v2);

  La representación gráfica del árbol que se crea es:

Figura 20. Representación gráfica del árbol de dependencias



- *Node.java*: Esta clase representa cada uno de los nodos del arbol. Cada nodo de nuestro *Tree* tendrá la siguientes información:

    - Lista de sus nodos hijos, si los tiene.
    - Nombre de la vista o de la tabla que representa.
    - Un booleano que indica si es vista o tabla
    - La profundidad en la que se encuentra en el árbol.
    - El código SQL asociado
    - El estado del nodo. Los estados del nodo pueden ser:
        a) Dont`know: indica que el nodo no se ha procesado todavía por lo que no se sabe si es un nodo válido o no. En este caso se marca mediante una mano 
        b) Valid:: indica que es un nodo válido. Los nodos en este estado aparecen marcados con un símbolo 
        c) Non-valid: indica que es un nodo erróneo. El símbolo utilizado en esta ocasión es un signo de admiración 
        d) Buggy: indica que es un nodo crítico. Nodo crítico es aquel que tiene todos sus hijos validos. El usuario no puede cambiar el estado de ningún nodo a Buggy, sino que es el propio depurador el que lo hace cuando detecta que es un nodo crítico. En ese momento se muestra un mensaje avisando al usuario de que el error se ha detectado. El símbolo asociado en este caso es el siguiente: 

    Al crear el árbol inicialmente todos los nodos tienen el estado don´t know.

- *MyRenderer.java:* Se encarga de la asignación de los símbolos, explicados en el punto anterior, a los nodos a medida que se vayan definiendo sus estados.

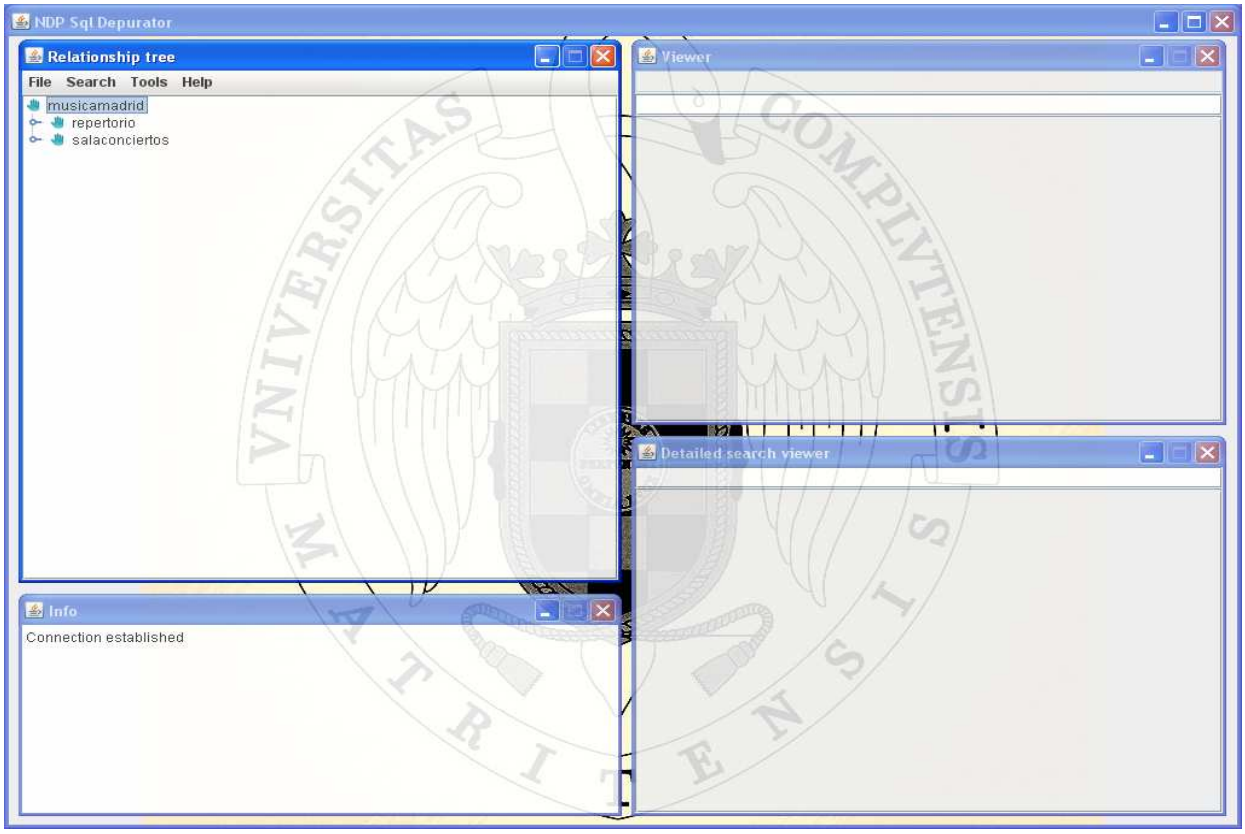
- *InstructionManager.java*: Esta clase es la clase que realiza el manejo total de las instrucciones, para ello hará uso de las clases ya mencionadas y también de la librería, ya mencionada, jsqlParser[10] que hemos incluido en el proyecto.

Después de que la herramienta realice todas estas acciones ya tendremos el archivo introducido por el usuario analizado, y una estructura de árbol que almacena toda la información necesaria para empezar con la depuración. Para realizar dicha depuración la herramienta necesita que el usuario vea el árbol de dependencias, y para crearlo gráficamente se utiliza la clase *JTree.java.*

- *JTree.java:* El JTree es el componente java visual que nos permite visualizar un árbol. En él podemos ver el típico árbol de datos en el que podemos abrir cada uno de los nodos para ver qué tiene dentro, cerrarlos, etc.

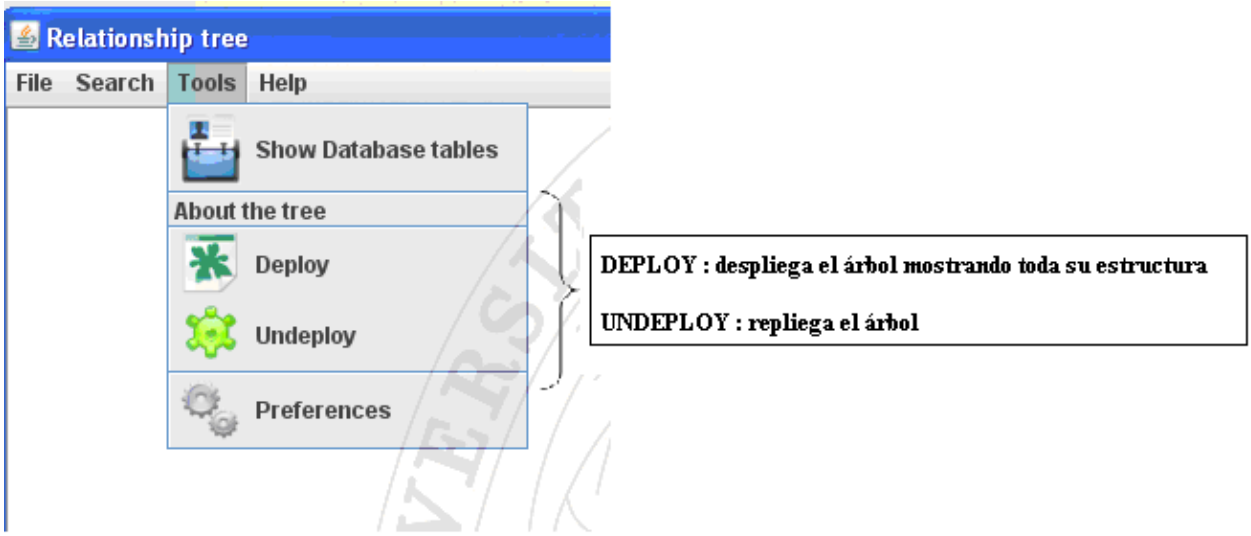
Una vez realizado definidamente el paso de la carga del archivo, se le muestra al usuario el árbol de dependencias en su respectiva ventana:

Figura 21. Inicio ejemplo de ejecución



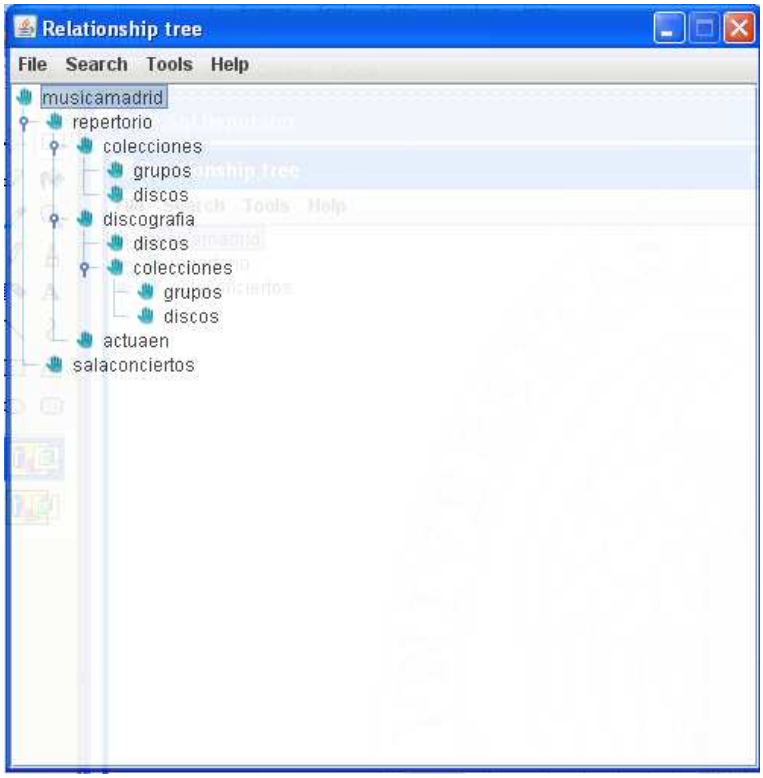
Para el fácil manejo del JTree en el menú **Tools** hay opciones que permiten desplegar o replegar el árbol de una sola vez evitando al usuario el tener que ir uno a uno.

Figura 22. Opción Deploy/Undeploy



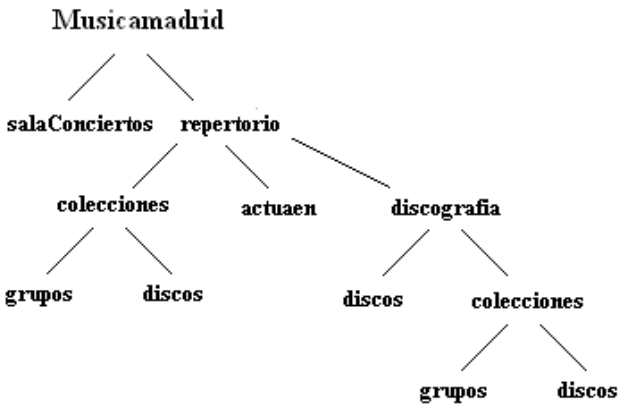
De tal forma que si el usuario pinchase sobre el la opción *Deploy* la ventana *Relationship tree* contendría el árbol de dependencias totalmente desplegado.

Figura 23. Ejemplo de despliegue del árbol sintáctico



Este sería el JTree obtenido de la estructura del árbol resultado de la clase *Tree.java:*

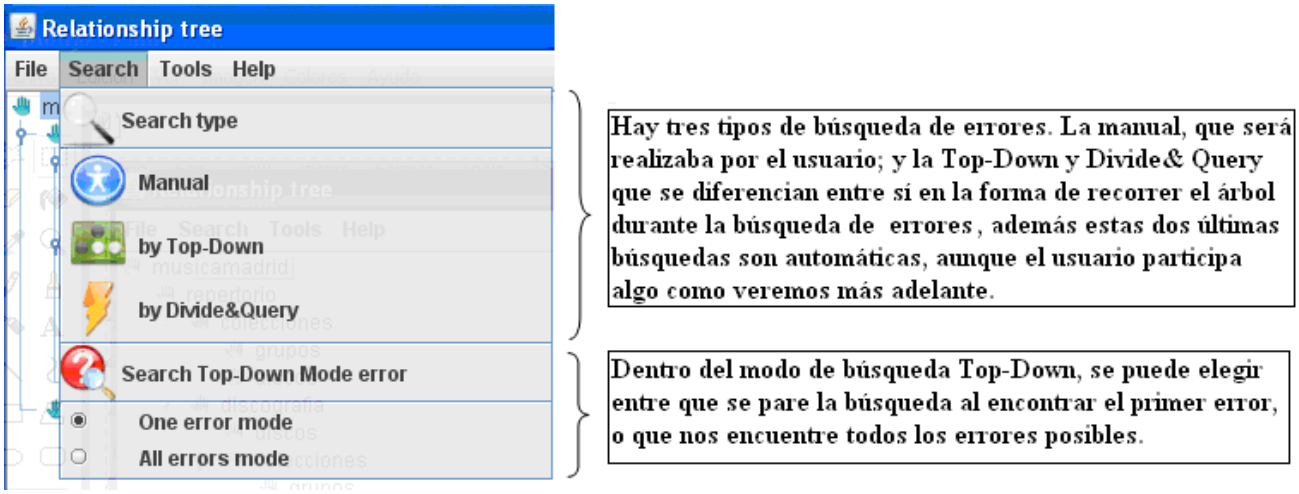
Figura 24. Estructura del JTree



- Una vez obtenido el árbol de cómputo, el depurador declarativo debe recorrerlo, investigando la validez de sus nodos hasta localizar un nodo crítico, en el caso de una vista; o un nodo erróneo, en el caso de una tabla.

Para empezar con la búsqueda el usuario puede utilizar cualquiera de las siguientes opciones que se presentan en el menú **Search:**

Figura 25. Opciones de búsqueda



De esto se encarga la clase *"view.java",* dicha clase representa a las vistas en el árbol de dependencias, por lo que en ella implementamos todo lo relacionado con éstas. Los métodos asociados a estas búsquedas son:

- public void lookForWrongNodes_Node (Node tn): Se encarga de la búsqueda by Top-Down.
- public void lookForWrongNodes_Log (Node tn): Se encarga de la búsqueda by Divide&Query.

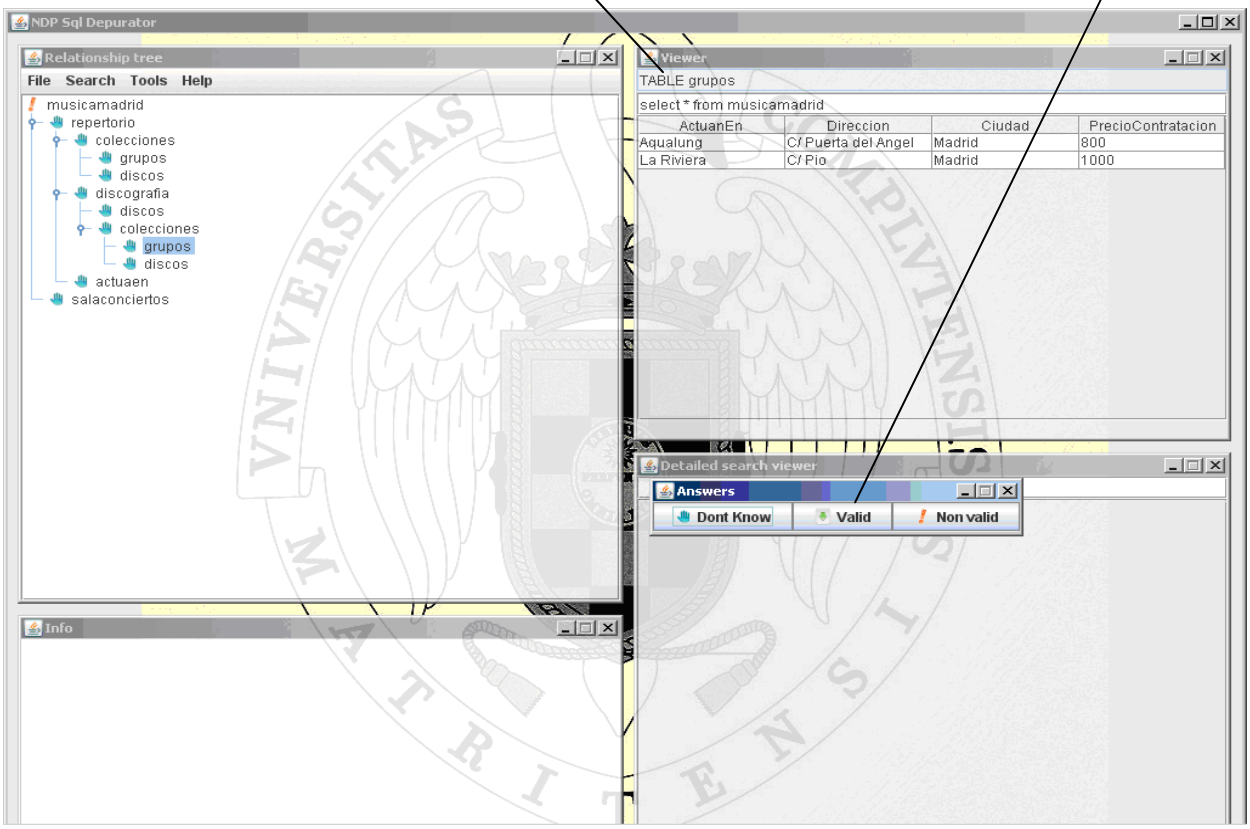
En este paso vamos a mostrar dos ejecuciones en las que se visualicen dos de las tres búsquedas diferentes que puede elegir el usuario:

1. El usuario escoge la opción **Manual :**

   Inicialmente se muestran todos los nodos del árbol en estado *dont-know,* excepto la raíz que tendrá un estado *non valid* ya que se corresponderá con la vista principal del archivo y, como es lógico, será errónea porque sino no se habría empezado elproceso de depuración. Además se mostrará el código asociado a la raíz o vista principal en la ventana **Viewer**.

   La herramienta pone a disposición del usuario una paleta de botones *"Answer"* mediante la cual se podrán cambiar los estados de los nodos.

Figura 26. Búsqueda manual



Este tipo de búsqueda se caracteriza en que el usuario puede ir mirando cada nodo del árbol en el orden que quiera, comprobando si los datos que contiene son los esperados, visualizándolos en la ventana **Viewer.** Si los datos mostrados son los esperados etiquetará al nodo como *Valid,* en caso contrario, se etiquetará como *Non Valid; e*n caso de duda se dejará o marcará como *Don´t know.*

En el momento en el que el usuario marque un nodo como *non valid* pueden pasar 2 cosas:

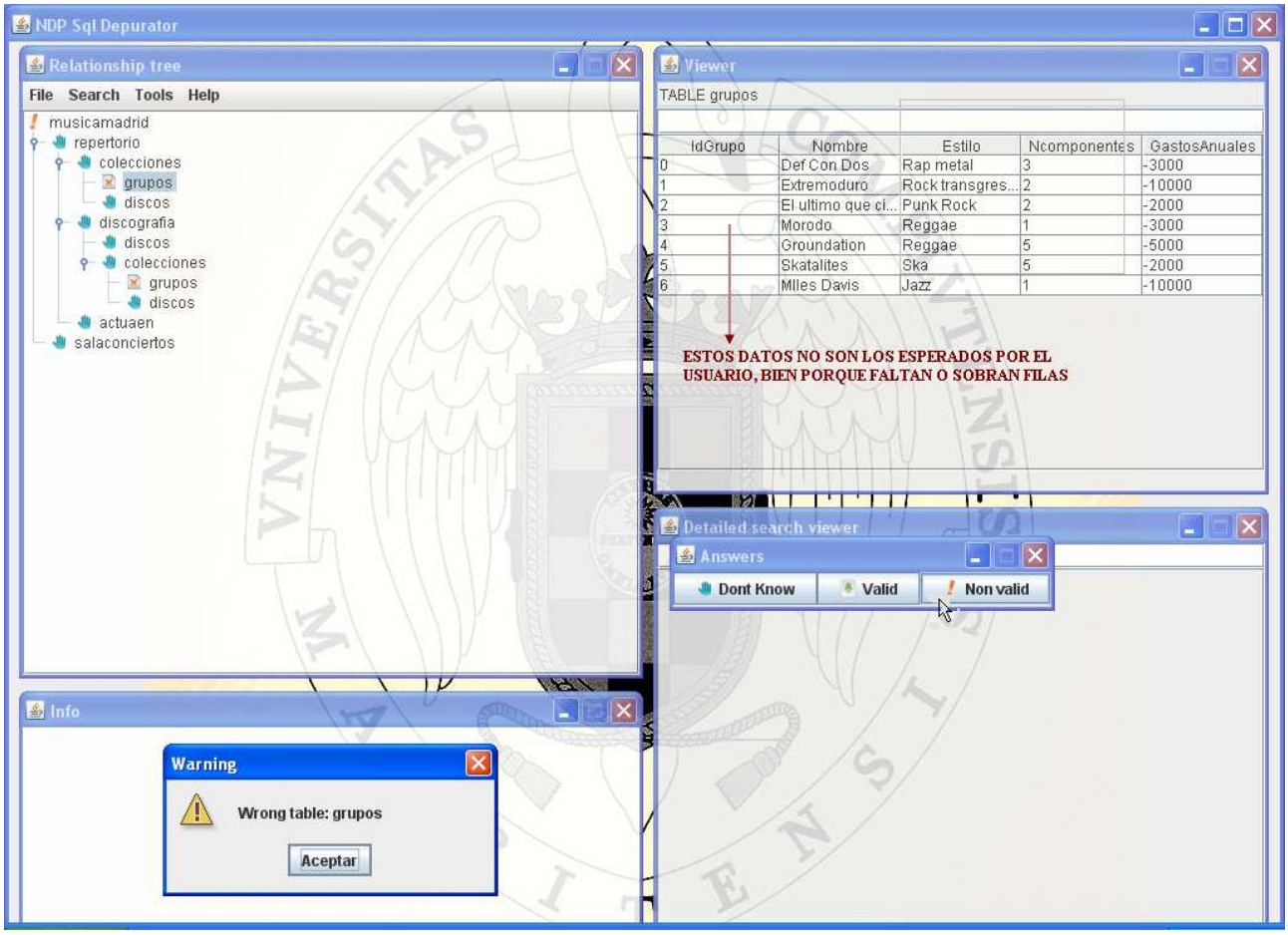
a) El nodo etiquetado es un nodo hoja.

b) El nodo etiquetado es un nodo intermedio.

a) El nodo etiquetado es un nodo hoja, por lo tanto, es una tabla. Esto quiere decir que el conjunto de datos almacenados en ella no son los que esperaba el usuario, entonces el depurador indicará mediante un mensaje que el error esta en la tabla respectiva.

En el ejemplo tacharemos como *non valid* a la hoja "grupos". A continuación, podemos observar la reacción del depurador:

Figura 27. Nodo hoja erróneo-Búsqueda manual

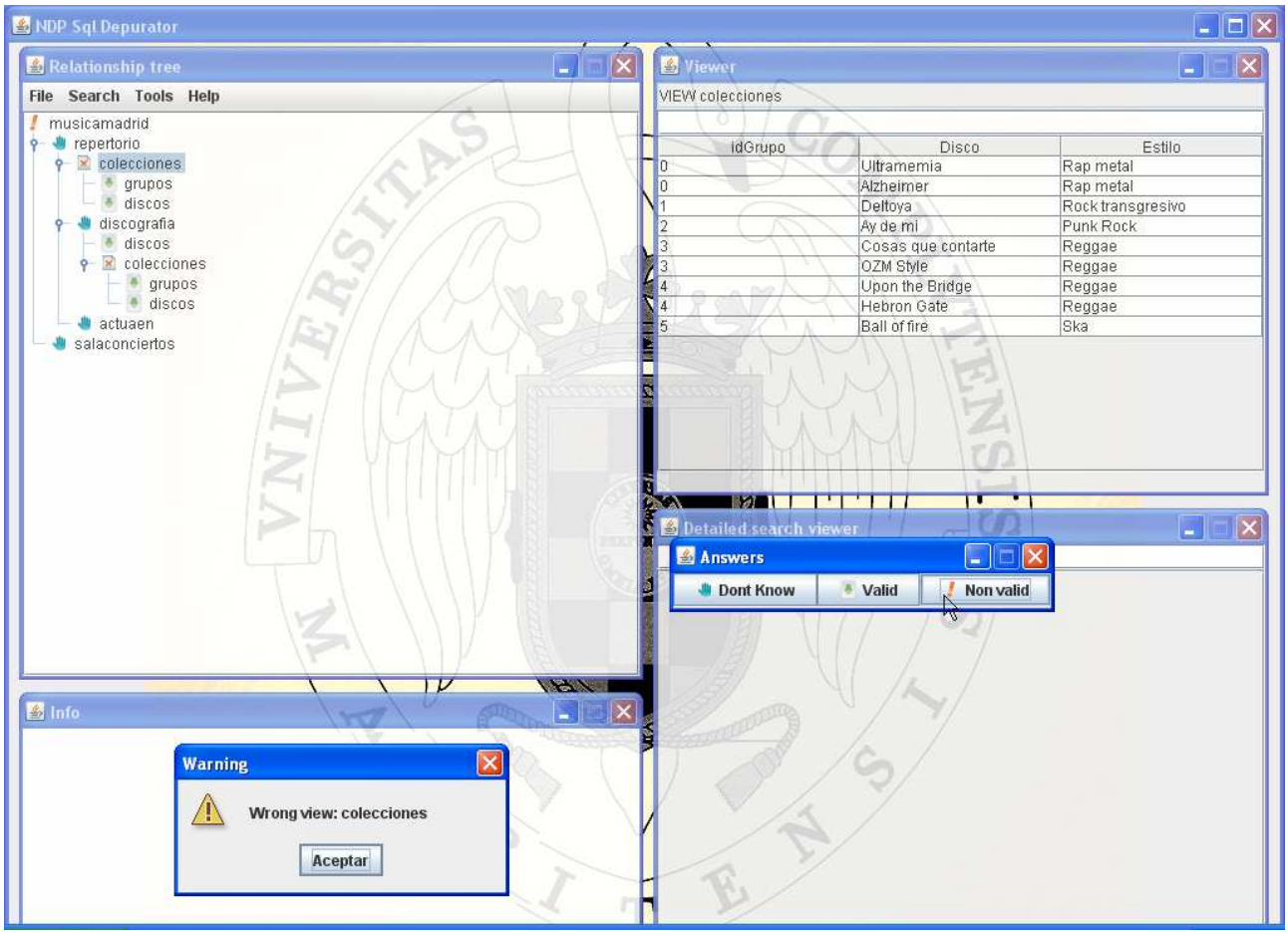


b) El nodo etiquetado es un nodo intermedio, con hijos, por lo que será una vista. Si el usuario ya ha mirado todos sus hijos y los ha etiquetado todos como *Valid*, entonces este, etiquetado como *non valid,* es un nodo crítico.

Esto quiere decir que el error se encuentra en el código asociado a dicho nodo, se señalará error, ofreciéndole al usuario, cuando sea posible, afinarlo.

En la imagen se muestra un ejemplo en el que el nodo intermedio "colecciones" es un nodo crítico:

Figura 28. Detectado nodo crítico-Búsqueda manual



Como se puede observar en las dos imágenes anteriores, cuando un nodo se marca con un estado se refleja en todos los nodos del árbol con el mismo nombre.

2. Volviendo atrás, en el caso de que el usuario no escoja la búsqueda **Manual**, podrá elegir entre **Top-Down** o **Divide&Query**.

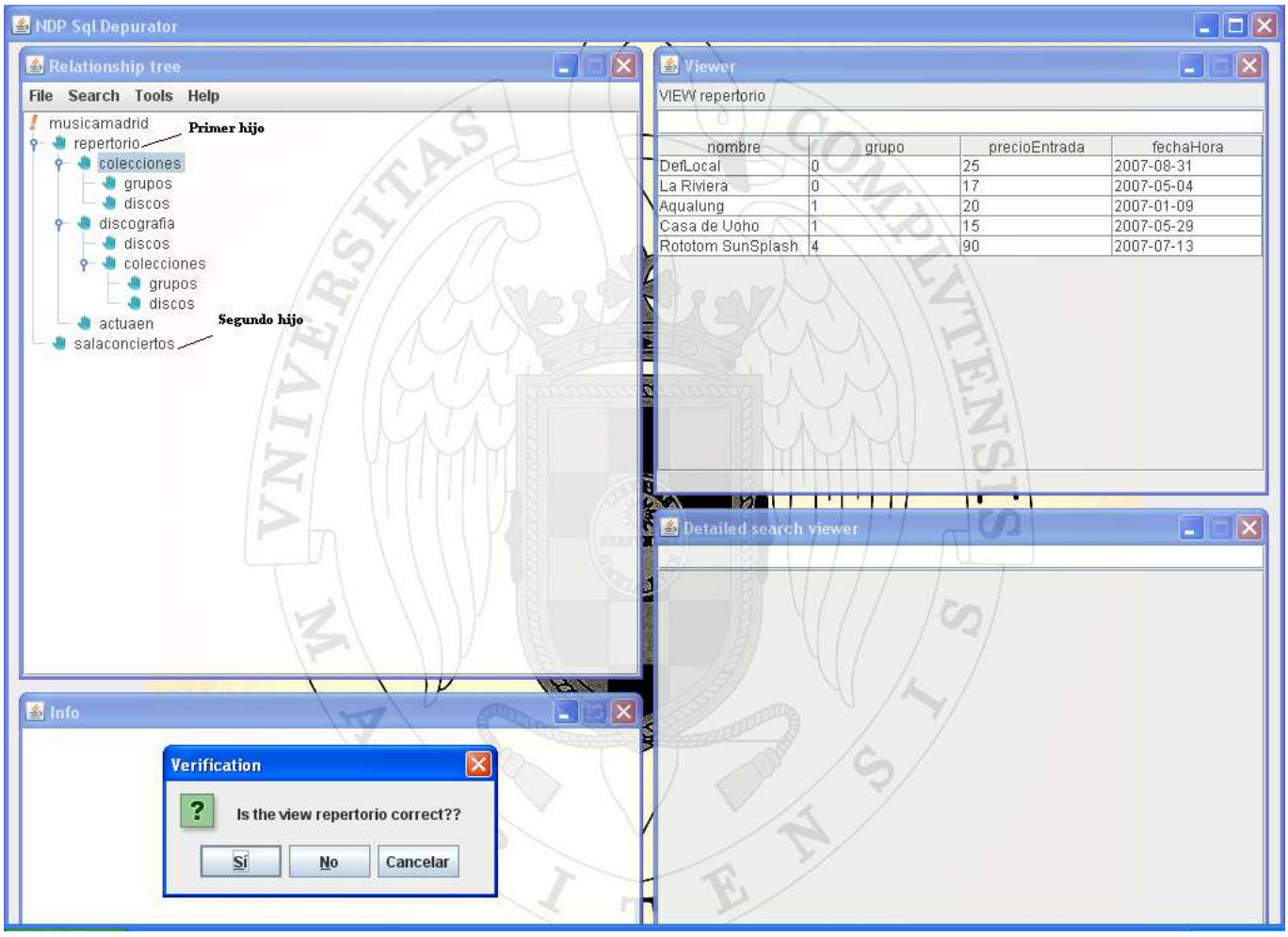
Cualquiera de estas dos búsquedas se hace con ayuda del usuario, al que el depurador va preguntado acerca de la validez de los nodos del árbol.

La única diferencia entre los dos métodos de búsqueda es el recorrido que realiza el depurador sobre el árbol. Dichos recorridos serán explicados en más detalle en el capitulo sobre *Posibles métodos de navegación*, ahora nos centramos en el uso de la aplicación para la búsqueda del error, no en el algoritmo en sí.

En el ejemplo vamos a elegir la búsqueda Top-Down, añadiéndole la opción de que se detenga al detectar el primer error. En este caso no habrá una paleta de botones para que utilice el usuario sino que los nodos los irá etiquetando el depurador según las respuestas que dé el usuario a sus preguntas.

Siguiendo el algoritmo, la primera pregunta se realiza sobre el primer hijo del nodo raíz, en el ejemplo la raíz es *musicamadrid* y su primer hijo es *repertorio,* una vista que tiene nodos hijo.

Figura 29. Búsqueda Top-Down



El usuario se vale de la ventana **Viewer** para ver si los datos son los esperados o no. Dependiendo de las respuestas del usuario y siguiendo el algoritmo el depurador realizará un recorrido u otro.

- Los datos son los esperados : el usuario responde **Sí**

Figura 30. Preguntas al usuario (1)

Al ser correcto se le etiqueta como *Valid* y se pasaría a preguntar sobre el segundo hijo *salaconciertos*, mostrando sus respectivos datos en la vista **Viewer:**

Figura 31. Estado de ejecución en búsqueda Top-Down (1)

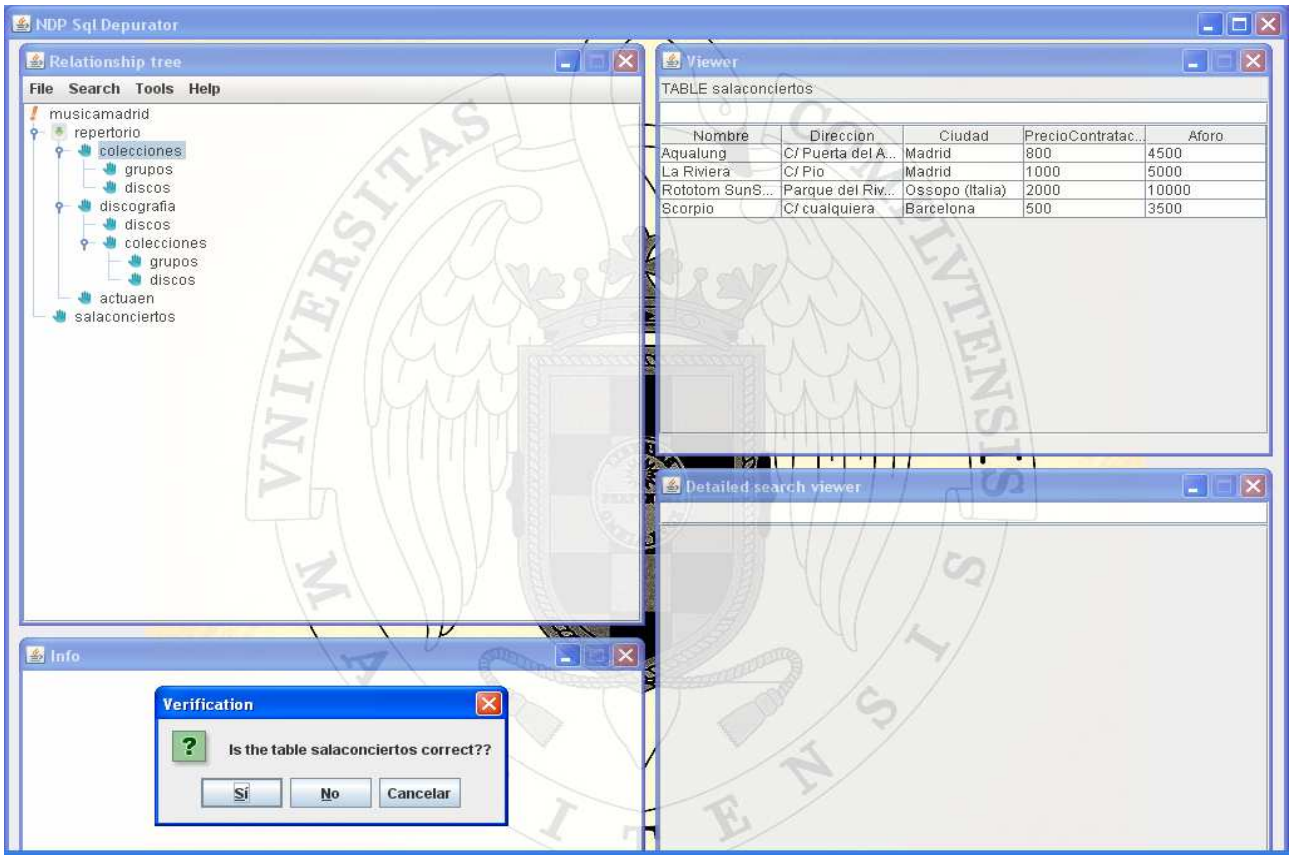
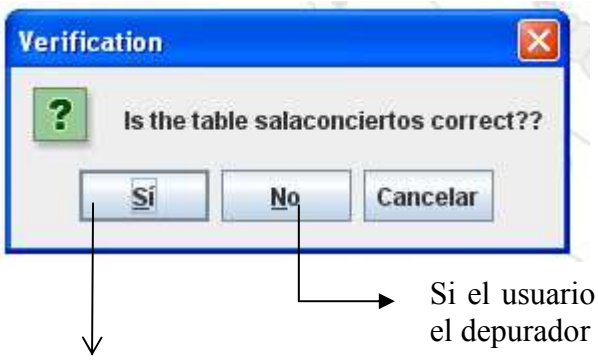


Figura 32. Preguntas al usuario (2)



Si el usuario responde **NO,** al ser el nodo una tabla el depurador habría encontrado el error.

Si el usuario responde **SÍ,** el depurador no puede seguir preguntando porque *musicamadrid* no tiene más hijos. Teniendo en cuenta el algoritmo seguido cuando llegamos a un nodo que es no válido y todos sus hijos válidos, se etiqueta como nodo crítico, señalando donde está el error:

Figura 33. Nodo crítico encontrado-Error detectado

Al ser una vista, el error se encuentra en el código asociado a dicho nodo, para localizar el error de manera más aproximada nuestra herramienta ofrece al usuario la opción de afinar la búsqueda. Pero como ya hemos mencionado en la búsqueda manual, este tema lo trataremos más adelante.

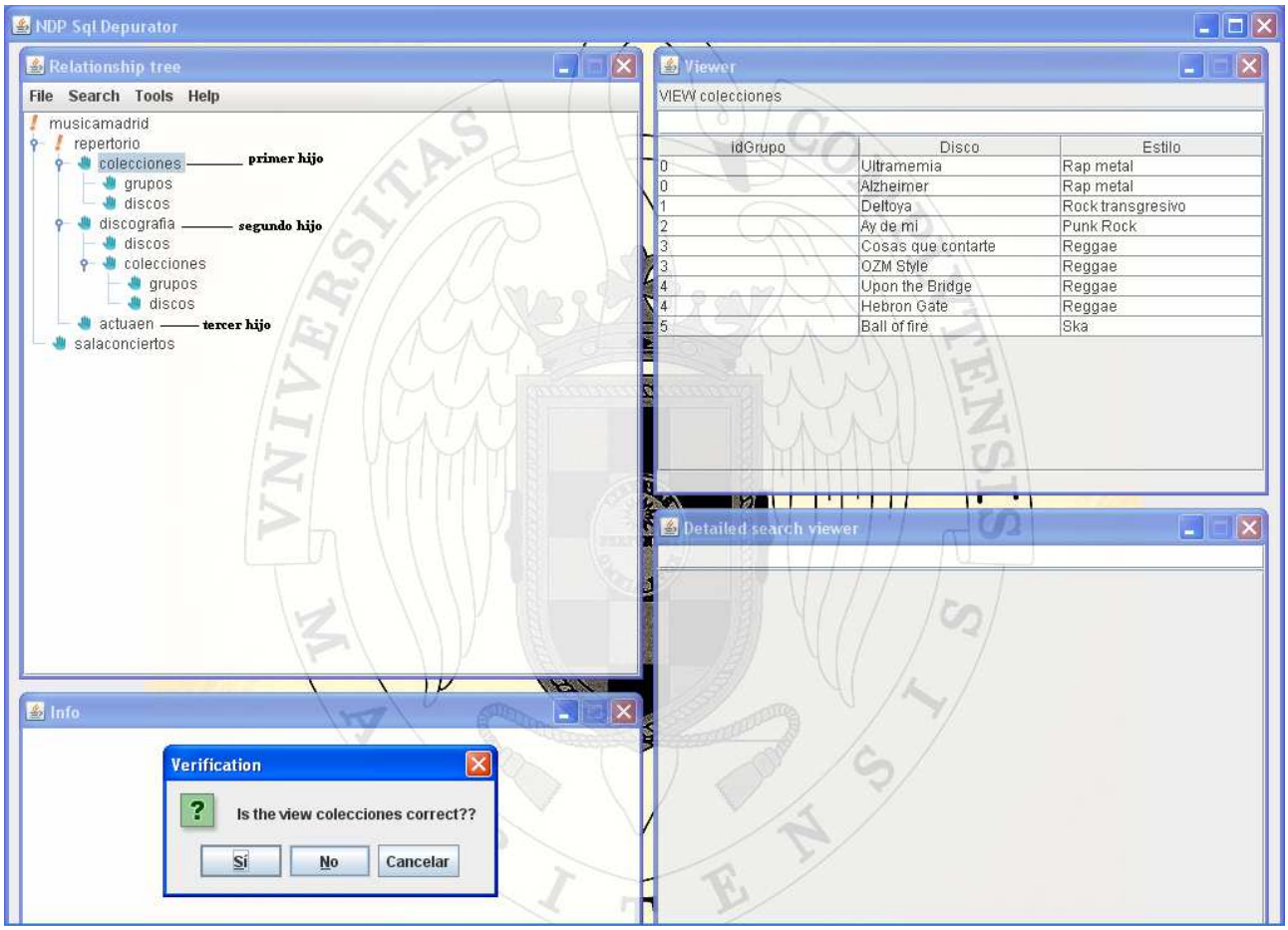
Volviendo atrás en el ejemplo, cuando el depurador pregunta sobre el primer hijo, *repertorio,* si en vez de estar de acuerdo con los datos, estos no fueran los esperados, la ejecución continuaría así:

- Los datos no son los esperados: el usuario responde **NO.**

Figura 34. Preguntas al usuario (3)

Es incorrecto, entonces se le etiqueta como *Non Valid.* Al tratarse de una vista y tener esta más hijos, según el algoritmo seguido se continuando aplicando el algoritmo sobre este subárbol, cuya raíz es *repertorio*. A continuación se preguntaría por el primer hijo, que se llama *colecciones.*

Figura 35. Final búsqueda Top-Down



Vamos a suponer que el usuario respondiese **NO,** ya que si fuese al revés el depurador pasaría a preguntar por el siguiente nodo hijo de *repertorio*, y el recorrido sería el mismo que en la primera llamada a la búsqueda cuando los datos de *repertorio* eran los esperados.

Al no ser los datos de *colecciones* los esperados**,** y al ser este una vista, el depurador preguntaría al usuario por sus respectivos hijos, además de etiquetar al nodo como *Non valid.* Este paso se corresponderá con el último de la búsqueda ya que estos hijos son hojas y pueden pasar, como hemos explicado anteriormente en la primera llamada, dos cosas:

a) El depurador preguntará por ambos hijos y el usuario contesta que son correctos en el siguiente orden:

Figura 36. Preguntas al usuario (4)

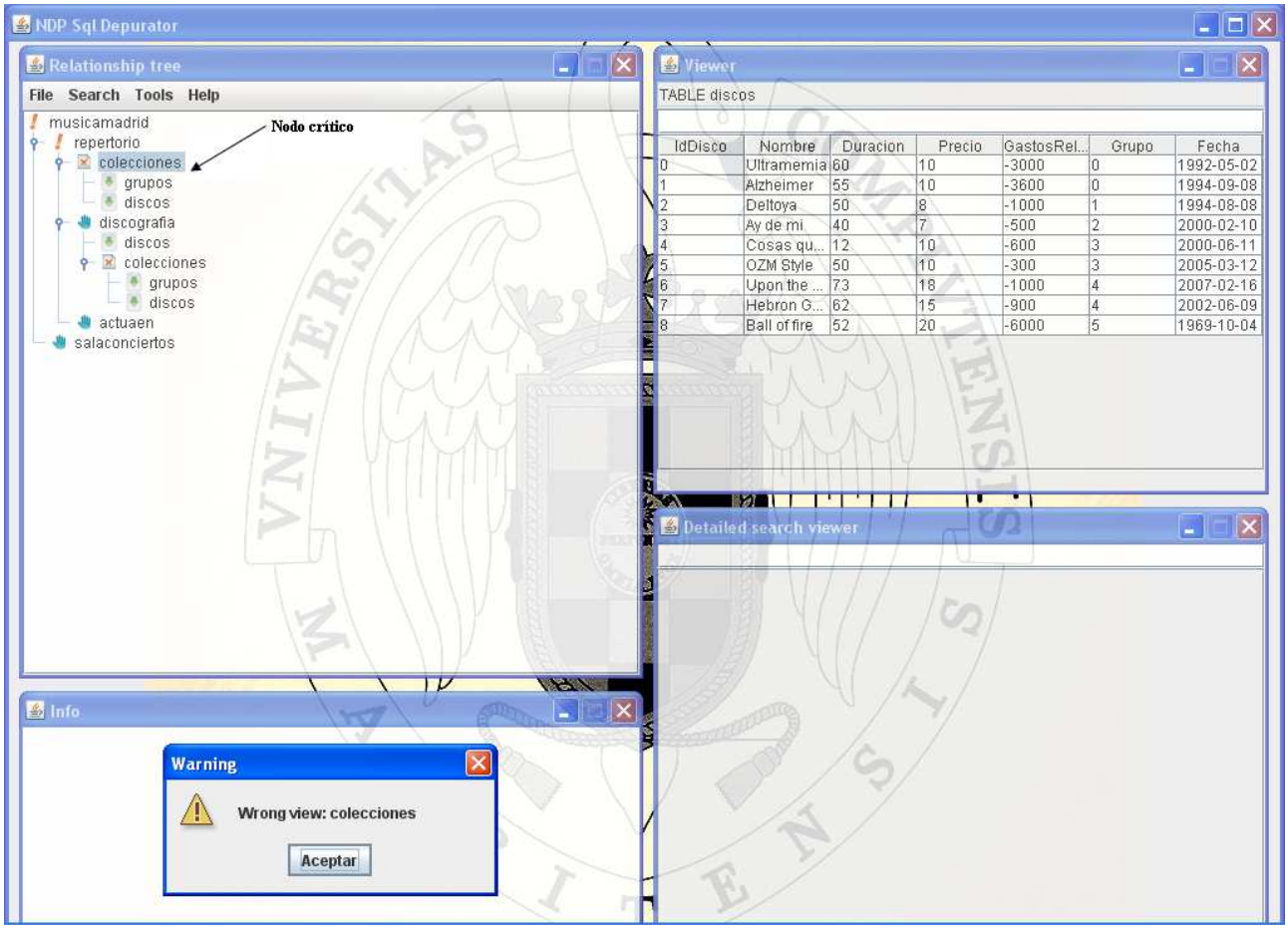
En el árbol de dependencias quedaría que el nodo *colecciones* es crítico por tener todos sus hijos válidos y el algoritmo pararía.

b) El depurador pregunta por los hijos y el usuario responde que alguno de ellos no es correcto. Al ser cualquiera de ellos hojas, el nodo representa una tabla, por lo que el error estará en los datos almacenados por ella. Esto será indicado mediante un mensaje de error por nuestro depurador y el algoritmo pararía.

- Como último punto de este capitulo vamos a hablar de la ayuda que ofrece nuestro depurador para hallar, una vez encontrado el nodo crítico, lo más aproximado posible el fragmento de código erróneo asociado al nodo crítico. Para ello vamos a realizar varios ejemplos

1. Continuamos el ejemplo anteriormente descrito sobre la búsqueda **Top-Down.** Recordamos como está la aplicación en el momento que se ha detectado el nodo *colecciones* como crítico:

Figura 37. Encontrado nodo crítico en búsqueda Top-Down



Llegado a este punto el usuario sabrá que el error se encuentra en la vista colecciones, sin embargo no sabe exactamente en que fragmento de código, es por eso que el depurador le pregunta si desea afinar el error:

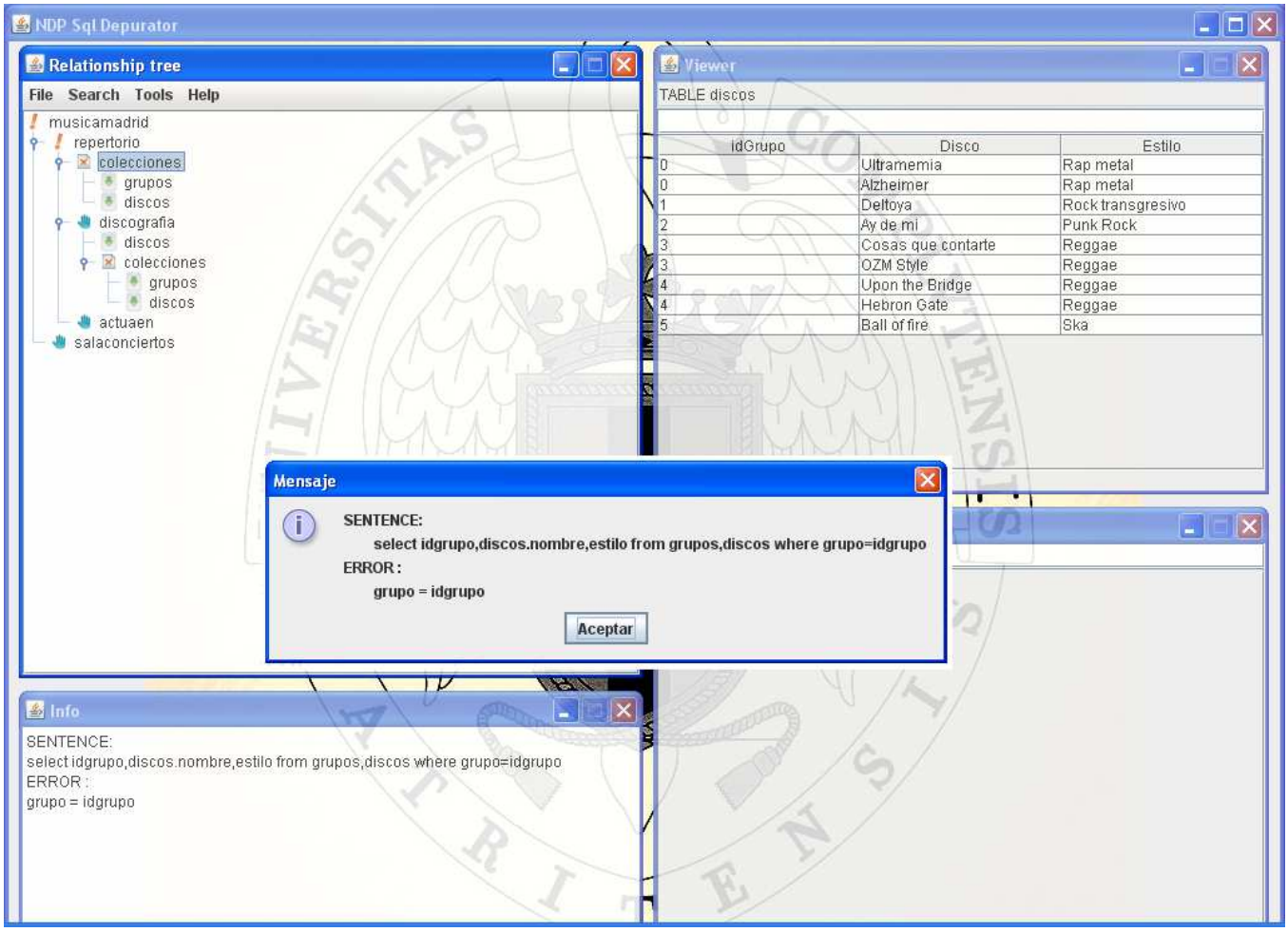
Figura 38. Posibilidad de realizar la búsqueda fina.

El código asociado a este nodo es el siguiente:

```
Create view Colecciones (idGrupo,Disco,Estilo) as
        (select idGrupo,Discos.Nombre,estilo
         from Grupos,discos
         where Grupo=idGrupo );
```

En este caso la búsqueda es muy poco compleja debido a que el código asociado a la vista es una sentencia SQL ***SELECT*** que contiene un ***WHERE*** con una sola condición, por lo que el error estará en dicha condición. A continuación podemos observar como el depurador indica el fragmento de código erróneo:

Figura 39. Detectado el error en el fragmento asociado al código del nodo crítico



Además de este tipo de sentencia SQL donde el WHERE solo tiene una condición, se podrían dar más casos:

1. El código asociado a la vista es un SELECT sin WHERE. En este caso nuestro depurador no podrá ayudar en la búsqueda aproximada del error. Esto es debido a que la parte del **WHERE** hace de filtro debido a sus condiciones, si estas no existen el error no proviene de aquí sino de las tablas del **FROM**.

2. El código asociado a la vista es un SELECT con un WHERE más complejo, donde las condiciones son conjunciones y/o disyunciones. Siguiendo el ejemplo, éste sería el caso de la vista musicamadrid, cuyo código asociado es el siguiente:

```
create view   MusicaMadrid (ActuanEn, Direccion,Ciudad,PrecioContratacion) AS
        (select Repertorio.nombre,Direccion,Ciudad,PrecioContratacion
         from Repertorio,SalaConciertos
         where Repertorio.nombre=SalaConciertos.nombre
                  and
                 (Ciudad='Madrid' or Ciudad='Barcelona'));
```

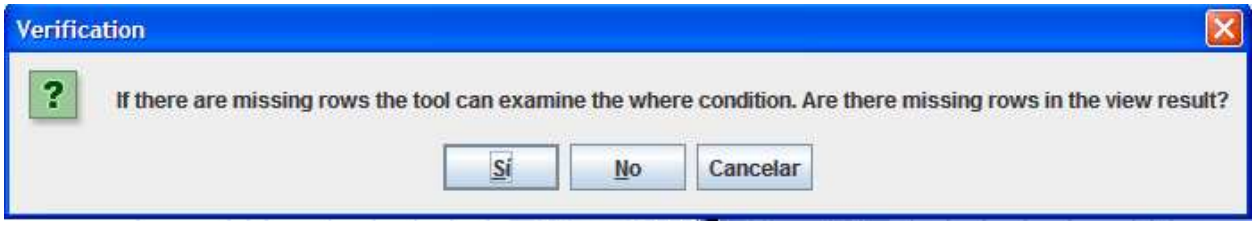
A continuación representaremos mediante una ejecución guiada este caso, donde se detecta al nodo *musicamadrid* como nodo crítico, pero antes explicaremos el algoritmo en el que se basa la búsqueda fina.

Como hemos indicado, para realizar la búsqueda fina, se parte de un buggy-node que ya hemos encontrado. Si es una tabla hemos acabado. Si es una vista se examina el WHERE, distinguiendo casos. Esta distinción de casos se hará en el método *check (Node checkNode)* de la clase *view.java.* La distinción es la siguiente:

1. Si no es si AND ni OR no se hace nada, como ya hemos explicado.

2. Si la condición C del WHERE es de la forma C= C1 AND C2 se muestra un cuadro preguntando:

Figura 40. Preguntas al usuario en caso de AND



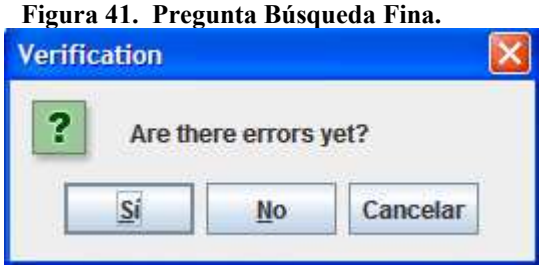
Si se dice que SI se hace una llamada al método *detailedSearch,* que es en el que se basa principalmente la búsqueda fina, de la clase

InstructionManager.java. Esta clase, como su propio nombre indica, se encarga del manejo de las instrucciones. Este método realiza la siguiente tarea:

2.1. Si C no es un AND se señala C como fuente del error y se acaba (en la primera llamada no puede ocurrir, pero como es recursivo en algún momento ocurrirá).

2.2 Si sí es un AND es de la forma C= C1 AND C2. Entonces se muestra en la ventana VIEWER el resultado de la vista sólo con C1 en el WHERE y se pregunta:

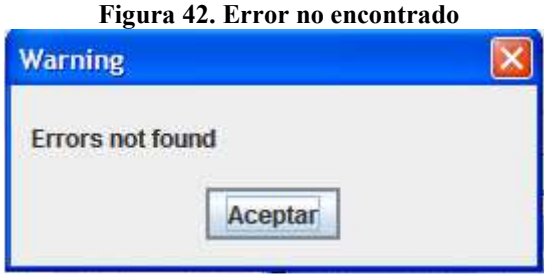
Figura 41. Pregunta Búsqueda Fina.



Si la respuesta es NO, entonces C2 es la condición que pierde las filas, es decir, el error se encuentra en dicha condición, y por lo tanto, llamamos recursivamente a detailedSearch con C2.

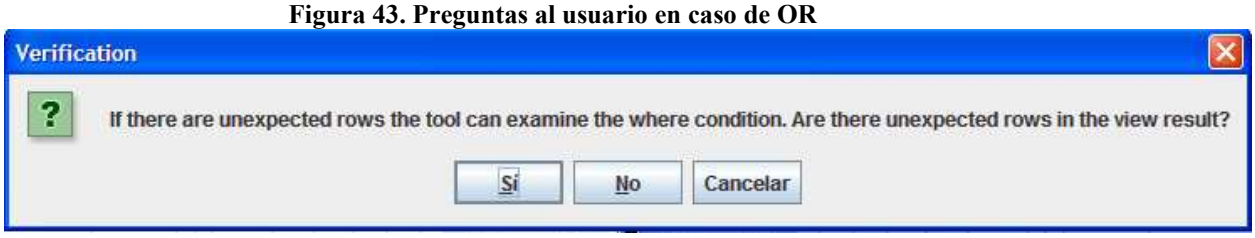
Si por el contrario la respuesta es SI, esto quiere decir que todavía faltan filas, entonces se muestra en VIEWER el resultado de la vista con C2 en el WHERE y preguntamos sin faltan filas. Si no faltan C1 es la causa de la pérdida de filas y, por lo tanto, llamamos a detailedSearch con C1. Si por el contrario siguen faltando, la culpa no está en el WHERE y paramos con un mensaje de aviso en el que indicamos que no hemos podido encontrar los errores:

Figura 42. Error no encontrado



3. Si la condición C del WHERE es de la forma C = C1 OR C2, entonces se pregunta:

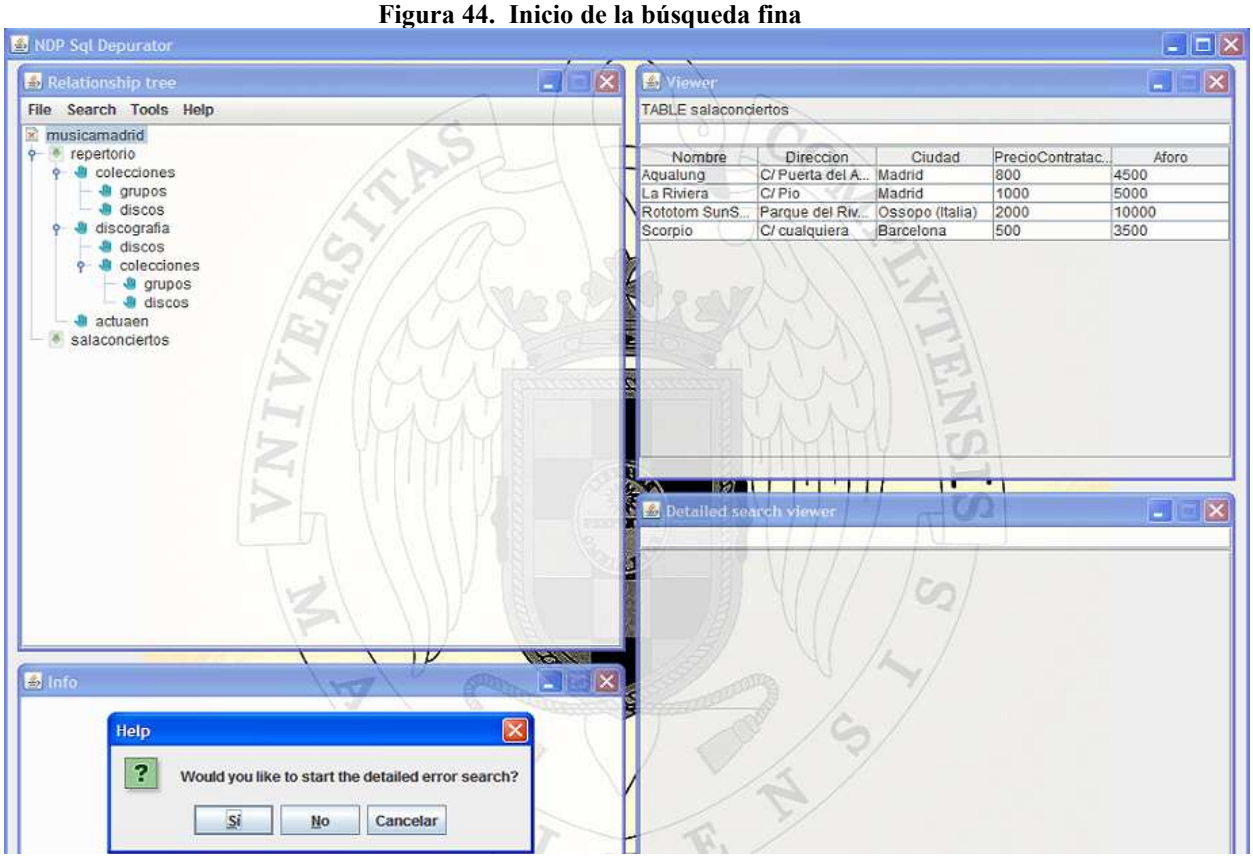
Figura 43. Preguntas al usuario en caso de OR



Y se procede igual pero en el procedimiento recursivo se busca la condición culpable de que sobren filas en lugar de que falten.

Ahora siguiendo el ejemplo de la memoria, presentamos la ejecución en el momento en el que tenemos el nodo *musicamadrid* como nodo crítico y la herramienta ofrece la opción de realizar la búsqueda fina:

Figura 44. Inicio de la búsqueda fina

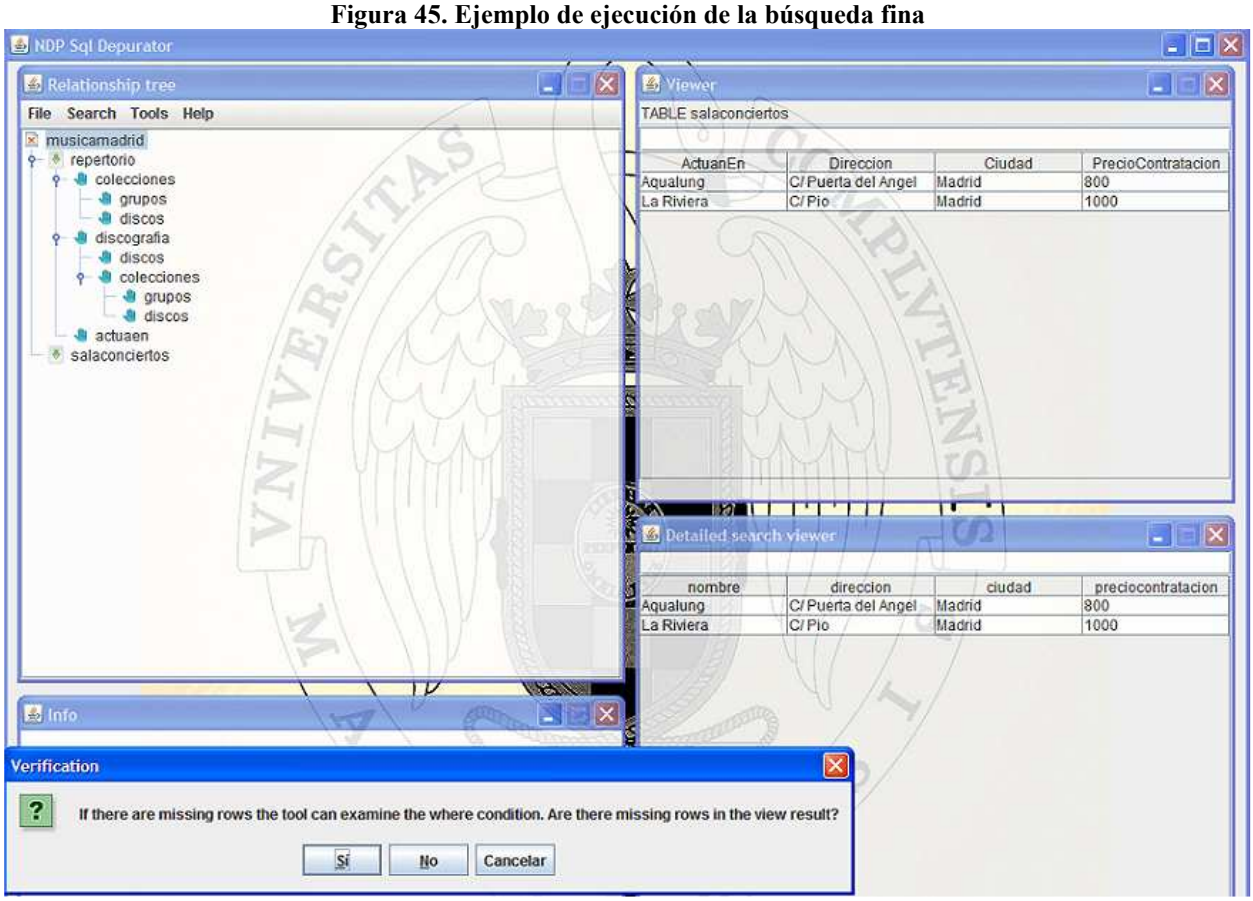


Comenzamos entonces a afinar el error, intentando aproximarlo lo máximo posible. En primer lugar, recordamos el código asociado a esta vista:

```
create view   MusicaMadrid (ActuanEn, Direccion,Ciudad,PrecioContratacion) AS
          (select Repertorio.nombre,Direccion,Ciudad,PrecioContratacion
           from Repertorio,SalaConciertos
           where Repertorio.nombre=SalaConciertos.nombre
                    and
                   (Ciudad='Madrid' or Ciudad='Barcelona'));
```

Al ser una expresión del tipo C1 AND C2, la herramienta nos preguntará si faltan filas, a lo que responderemos afirmativamente:

Figura 45. Ejemplo de ejecución de la búsqueda fina

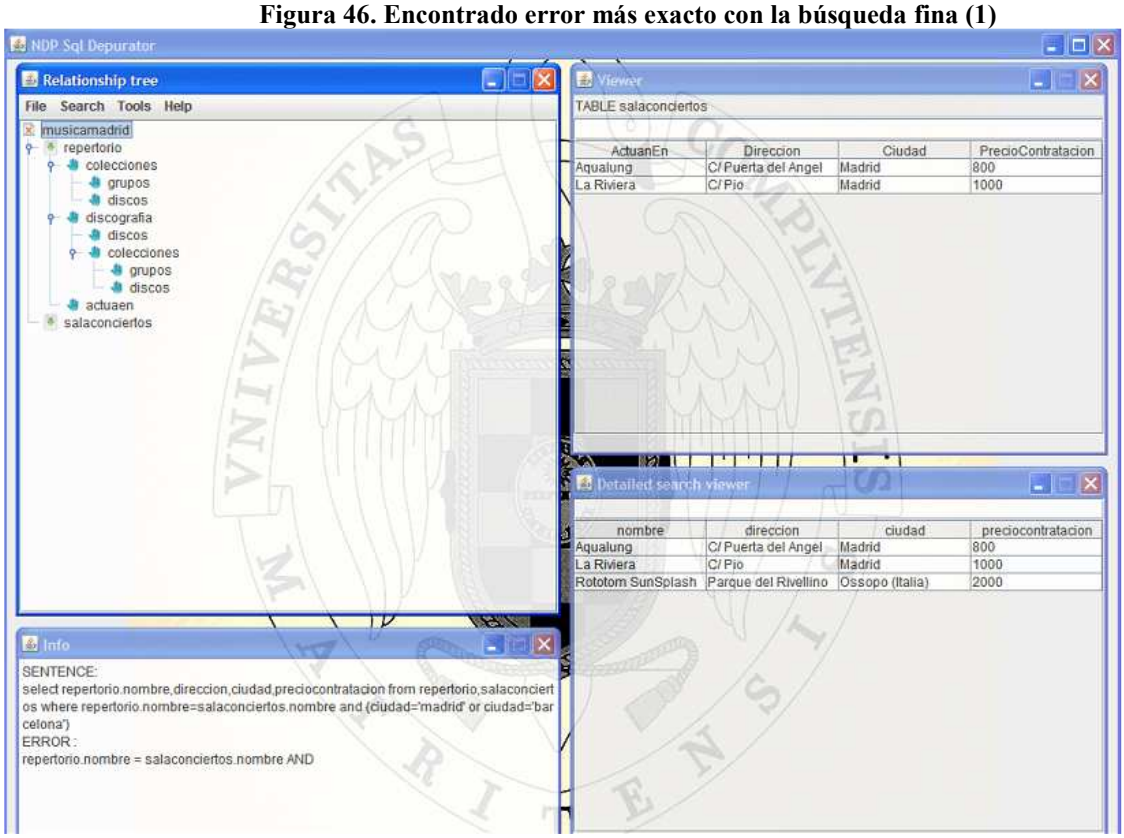


Al haber afirmado que faltan filas, se mostrará, siguiendo el algoritmo, en la ventana VIEWER, el resultado de la vista poniendo en el WHERE sólo la parte izquierda de la AND, de forma que el código asociado a la vista *musicamadrid* quedaría:

```
create view   MusicaMadrid (ActuanEn, Direccion,Ciudad,PrecioContratacion) AS
        (select Repertorio.nombre,Direccion,Ciudad,PrecioContratacion
         from Repertorio,SalaConciertos
         where Repertorio.nombre=SalaConciertos.nombre);
```

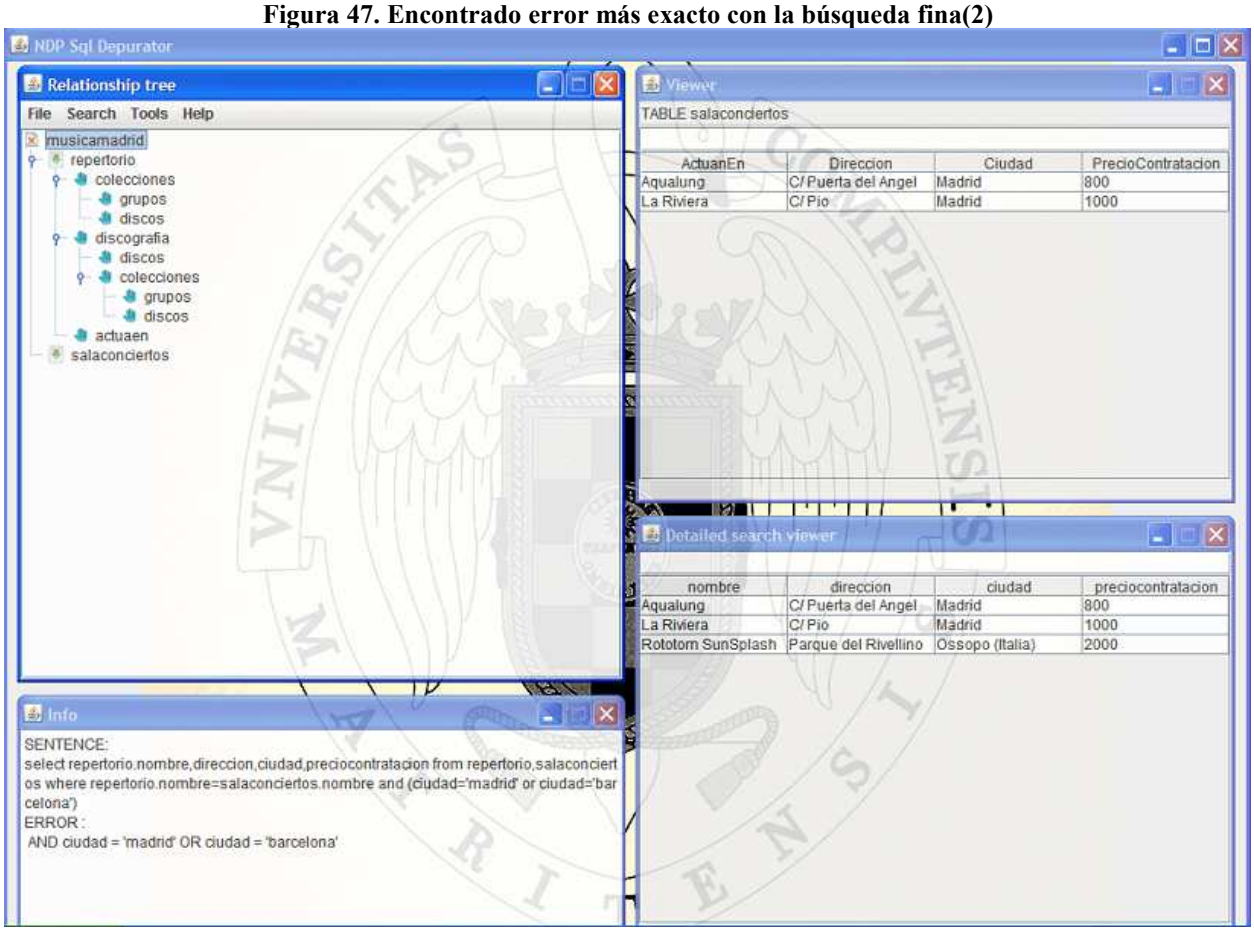
Nuestro depurador nos preguntará si continúan faltando filas, si es así y sigue habiendo pérdidas de filas, el depurador nos devolverá que el error está en esta condición. Al ser una condición simple, la llamada recursiva cesa y el depurador nos devuelve donde se encuentra el error. La imagen de la herramienta en este punto la podemos observar a continuación.

Figura 46. Encontrado error más exacto con la búsqueda fina (1)



Si por el contrario no faltan filas, esto quiere decir que el error se encuentra en la otra condición, que se corresponde con: “and (Ciudad='Madrid' or Ciudad='Barcelona')”. Aunque no se trata de una condición simple el depurador no hace llamada recursiva ya que se trata de una OR, por lo tanto para aquí, quedando la aplicación de la siguiente forma:

Figura 47. Encontrado error más exacto con la búsqueda fina(2)



Así, mediante este algoritmo de búsqueda afinada, no sólo conseguimos encontrar la tabla o vista errónea, sino que además si se trata de una vista, somos capaces de localizar el fragmento de código asociado a esto incorrecto de la forma más aproximada posible.

3.2 Problemas encontrados.

A continuación, comentaremos los distintos problemas que hemos tenido a lo largo de la implementación de la aplicación.

En un principio nuestro primer problema era como parsear las instrucciones de MySQL para traducirlas y manejarlas en Java, para ello barajamos las siguientes posibilidades:

1.- Programar nuestro propio parser desde cero, esta tarea supondría un trabajo bastante complicado así que descartamos esta opción desde un principio.

2.- Hacer nuestro propio parser utilizando un generador como JavaCC , ANTLR o LPG. Esta opción también nos resultó bastante costosa, debido a la complejidad de dichos generadores, además, teniendo en cuenta que nuestro proyecto no se basaba solamente en esta tarea decidimos buscar otra opción.

3.- Buscar una gramática ya creada y utilizar el generador, pero el hecho de encontrar una gramática completa para SQL estándar no es tan fácil. Encontramos múltiples para subconjuntos o implementaciones concretas pero la gran gramática que buscábamos es esquiva y se oculta bien.

4.- Basarnos en un parser ya existente y de software libre. Esta opción fue la escogida finalmente, empezando así nuestra búsqueda de parsers ya construidos.

Dimos con distintas alternativas:

- **ZQL**: basado en un parser generado con JavaCC. Descartado por su falta de garantías y porque la batería "exhaustiva" de pruebas que muestra deja mucho que desear.
- **SQL4J**: Parecía sencillo pero la falta de documentación y el fallo en ejemplos simples provocaron su descarte.
- **SQL development Tools Project** dentro de Eclipse: basado en LPG.
- **JSqlParser**: Proyecto de Software Libre alojado en SourceForge bajo licencia LGPL, basado en un parser generado con JavaCC.

  En un primer momento asusta su complejidad aparente pero vale la pena el esfuerzo. Además su documentación es aceptable, contribuyendo a esto la ayuda permanente de un foro que puede dar respuestas a nuestras preguntas.

  Es útil tanto para analizar como para generar consultas SQL, por todo esto fue la opción finalmente escogida.

Nuestro siguiente problema fue la implementación de los tres tipos de búsqueda de errores que posee nuestra aplicación, y que han sido explicados en profundidad anteriormente:

1.- Manual: Este en concreto no tuvo muchos problemas ya que se basa en un recorrido manual sobre el árbol de vistas y tablas.

Fue en los dos siguientes en los que tuvimos más complicaciones:

2.- By Top-Down: Aquí nuestro problema principal fue concretar la idea general del algoritmo y ponernos de acuerdo, una vez conseguido esto programarlo fue más fácil.

3.-By Divide & Query: Tuvimos que documentarnos sobre los algoritmos de Divide y Vencerás, concretamente en los basados en árboles.

Nuestra siguiente dificultad apareció una vez que ya nuestra herramienta localizaba la vista/tabla errónea. Decidimos ampliar las opciones de nuestra aplicación de tal forma que si el error se encontraba en una vista se ayudara al usuario a encontrar el error, de forma más exacta, dentro del código asociado a la vista/tabla errónea que ya devolvíamos. Nuestro principal obstáculo aquí fue a la hora de parsear la parte del where en las instrucciones SQL, ya que, debido a que será aquí donde estará normalmente el error, queríamos reducir al máximo su rango.

Esta parte de la sentencia a menudo estará formada por conjunciones y/o disyunciones (AND y OR) de distintas condiciones, por lo que queríamos devolver de manera más precisa la condición errónea. Si esto no fuera posible devolveríamos la conjunción o disyunción, más concreta posible, donde se encontrará el fallo.

3.3 Limitaciones de nuestro sistema.

En nuestra aplicación encontramos varias limitaciones relacionadas, principalmente, con el archivo SQL que queremos depurar.

En primer lugar este archivo debe ser un archivo con extensión txt, formado solo y exclusivamente por sentencias SQL.

Cada sentencia irá en una línea y terminará en ;

El archivo deberá contener una vista errónea principal, que será la que queremos depurar. La creación de esta vista será la última línea en el archivo. De ella colgarán las tablas y vistas de las que dependerá, formando el árbol de depuración. Dicho árbol será el que se recorrerá en busca de la tabla/vista errónea.

Con respecto al código de las sentencias, en nuestro depurador existen consultas que no se admiten, no porque no se puedan ejecutar, ya que nuestra aplicación ejecuta cualquier código SQL, sino debido a las limitaciones del JsqlParser[10].

Hemos reducido estas limitaciones añadiendo el comando SQL CREATE VIEW; JsqlParser no contenía este comando, totalmente necesario en nuestra aplicación, así que hemos añadido su parser y su manejo en Java. La sintaxis aceptada para la creación de una vista es la siguiente:

CREATE VIEW "NOMBRE _ VISTA" AS "Instrucción SQL";

La “Instrucción SQL” puede ser cualquiera de las instrucciones SQL que admite nuestra aplicación.

Además de esto hemos incluido mejoras en algunos comandos para ampliar el campo de las posibles sentencias que se pueden depurar, así, por ejemplo, en la instrucción SQL SELECT hemos mejorado el FROM; el jsqlparser solo permite el FROM de una tabla mientras que nosotros hemos incluido la opción de que se pueda hacer el FROM de varias tablas.

3.4 Extensiones posibles del programa.

Tenemos en mente diferentes extensiones posibles de la aplicación. A continuación numeraremos algunas.

En primer lugar, la idea sería aumentar la herramienta para fragmentos de SQL que no estén aún considerados, como es el caso de las uniones e intersecciones en la búsqueda fina.

Otra extensión sería integrar un Editor así el usuario podría modificar el código de manera on-line.

Podríamos mejorar la presentación de la aplicación consiguiendo una interfaz más llamativa y sorprendente.

Nos gustaría incluirlo como parte de una herramienta global que genere casos de prueba para probar las consultas y utilice nuestro depurador para encontrar los errores si los hubiera.

Utilizar prototipos para contestar las preguntas en lugar del usuario. Por ejemplo si se tiene una versión antigua o menos eficiente de la vista principal que funcionaba correctamente se le podría indicar al programa para que la usara y comparará con la respuesta que da el código que está depurando. Cuando las respuestas coincidan el nodo se supone válido y cuando no incorrecto. De esta forma se podría encontrar el error sin apenas hacer preguntas al usuario (sólo se harían preguntas para las vistas nuevas que estén en la versión que se está depurando y no en la antigua). Esta idea es fácil de implementar y muy potente, pero por falta de tiempo no ha podido llevarse a cabo.

Capitulo 4

POSIBLES MÉTODOS DE NAVEGACIÓN

Una vez obtenido el árbol de cómputo, el depurador declarativo debe recorrerlo, investigando la validez de sus nodos hasta localizar un nodo crítico. Esto se hace con ayuda del usuario, al que el depurador va preguntado acerca de la validez de los hechos básicos contenidos en los nodos del árbol. Llamamos *navegación* a este recorrido del árbol, y *estrategia de navegación* al algoritmo seguido por el depurador durante la navegación.

En nuestra herramienta existen tres métodos diferentes de navegación a través del árbol, que nos facilitan la búsqueda de errores:

- Manual
- By Top-Down
- By Divide & Query

Los dos últimos basados en estrategias de navegación que explicaremos detalladamente más adelante.

El uso de diferentes estrategias de navegación puede tener consecuencias sobre:

1. El número de preguntas realizadas al usuario y su complejidad. En efecto, uno de los mayores inconvenientes habitualmente asociados a la depuración declarativa es el gran número de preguntas que el depurador necesita realizar, así como su complejidad para el usuario. En este capítulo compararemos dos estrategias guiadas por el depurador: la estrategia *descendente (***By Top-Down***)* y la estrategia *pregunta y divide (***By Divide & Query**), mostrando sus características. También veremos las ventajas en este sentido que puede tener el permitir la inspección libre del árbol por parte del usuario utilizando las opciones a este efecto disponibles en el navegador.

2. La eficiencia del depurador. El otro inconveniente mayor de la depuración declarativa es el coste requerido, tanto en memoria como en número de preguntas al usuario, para la evaluación del árbol de cómputo. Algunas estrategias (en particular la estrategia *descendente*) permiten la navegación perezosa del árbol, evitando así su evaluación completa y mejorando la eficiencia en cuanto a consumo de recursos del depurador. Por desgracia la evaluación perezosa es incompatible con las técnicas que hasta ahora

conocemos para implementar la otra estrategia (la *pregunta y divide*) que resulta preferible en muchos casos para lograr una menor cantidad de preguntas al usuario (punto anterior) así como con algunas de las características más interesantes del depurador. Valoraremos mediante datos empíricos el coste de ambas posibilidades.

4.1 Método de navegación manual

Este método es totalmente interactivo, el usuario va chequeando los nodos que desea, marcándolos como non-valid (nodo inválido), valid (nodo válido) o dont-know (no se sabe si el nodo es válido o no).

Cuando se marque un nodo como non-valid, el cual corresponda a una tabla, la aplicación mostrará un mensaje indicando que esta tabla es errónea. Si no es así, es decir, no se marca ningún nodo que se corresponda con una tabla como non-valid, cuando se llegue un momento en el que un nodo asociado a una vista está marcado como non-valid y todos sus hijos en el árbol como valid, la aplicación mostrará esta como buggy (nodo crítico) y dándole al usuario, seguidamente, la opción de realizar una búsqueda fina del error.

4.2 Método de navegación By Top-Down

Este método navega el árbol siguiendo la siguiente estrategia:

- Algoritmo descendente:

Sea T un árbol y R = raíz (T), con R no válido.
- Si todos los hijos de R en T son válidos
- Parar, señalando a R como nodo crítico.
- En otro caso
- Seleccionar un hijo no válido N Є hijos (T, R)
- Repetir el proceso recursivamente sobre subárbol (T, N).

La siguiente proposición muestra que la navegación siguiendo esta estrategia conduce siempre a la localización de un nodo crítico. De aquí en adelante utilizaremos la notación $|T|$ para representar el número de nodos en un árbol *T*.

Proposición. *Sea T un árbol con raíz incorrecta. Entonces la estrategia de navegación descendente finaliza en un número finito de pasos, señalando a un nodo crítico de T.*

Demostración

En cada llamada recursiva la estrategia utiliza un nuevo árbol *T' = subárbol (T, N)*, con *N* ∈ *hijos (T, R)*, por lo que se verifica $0 < |T'| < |T|$, ya que $R \notin T'$ y al menos $N \in T'$. Esto garantiza que el algoritmo termina en un número finito de pasos porque aunque en ningún paso intermedio se encontrara con un nodo con todos los hijos válidos, la disminución del tamaño garantiza que se llegará finalmente a un árbol-hoja, que si cumple esta condición de parada.

Para comprobar que el nodo señalado es crítico en *T* podemos proceder por inducción sobre $|T|$. Si $|T| = 1$ el árbol es una hoja y al ser su raíz no válida ésta es un nodo crítico, tal y como señalará la estrategia. Si $|T| > 1$ y su raíz es crítica la estrategia también la señalaría correctamente sin efectuar ningún paso recursivo. En otro caso se procederá recursivamente sobre *T' = subárbol (T, N)* con *N* no válido. Al ser $|T'| < |T|$ y *raíz (T')* un nodo no válido la hipótesis de inducción de la estrategia localizará un nodo crítico en *T'* que, al ser *T'* subárbol de *T*, también será nodo crítico en *T*.

Hay que observar que implícitamente la validez de esta proposición depende de la corrección del test de validez, es decir de las respuestas del usuario. Si éste no responde adecuadamente a las preguntas del navegador la estrategia puede proporcionar resultados incorrectos (aunque la condición de terminación sí se mantiene).

4.3 Método de navegación By Divide & Query

La estrategia seguida en este caso es *pregunta y divide.* Al igual que en la estrategia descendente la depuración comienza con un árbol cuya raíz es no válida, propiedad que se conserva tras cada paso. La idea es seleccionar en cada paso un nodo *N* tal que coincida el número de nodos dentro y fuera del subárbol cuya raíz es *N*. Aunque este nodo, al que se llama el *centro* del árbol, no existe siempre, la estrategia busca el nodo que mejor aproxime esta condición. Para simplificar la explicación consideramos una función:

$$dif\,(T, N) = (|T| - |subárbol(T, N)|) - |subárbol(T, N)|$$

que calcula la diferencia entre el número de nodos fuera y dentro del subárbol cuya raíz es *N*. El nodo buscado es entonces aquel $N \in T$ tal que:

$$|dif(T, N)| = min\ \{|dif(T, M)| : M \in T\}$$

Si el nodo *N* que cumple esta propiedad es no válido su subárbol será considerado en el siguiente paso, mientras que si es válido dicho subárbol sería eliminado. Todas estas ideas quedan reflejadas en la siguiente descripción en pseudocódigo de la estrategia:

- Algoritmo pregunta y divide:

Sea T un árbol y R = raíz(T), con R no válido.
- Si |T| = 1
- Parar, señalando a R como nodo critico
- En otro caso
- Sea N ∈ T tal que |dif(T, N)| = min{|dif(T, M)| : M ∈ T}
- Si N es válido
- Sea T' = T - subárbol (T, N)
- Repetir el proceso sobre T'
- En otro caso
- Repetir el proceso sobre subárbol (T, N)

La operación *T – subárbol (T, N)* denota la eliminación de *T* de todos los nodos en *subárbol (T, N)* utilizando la operación de eliminación de la siguiente definición, que equivale a la eliminación del subárbol completo.

Definición. Sea *T* un árbol y *N ∈ T* un nodo tal que *N != raíz(T)*. Sea *M* el padre de *N* en *T* (es decir *N ∈ hijos* (*T, M*)). Entonces la notación *T - N* representará el nuevo árbol obtenido al eliminar *N* de *T*, dejando los subárboles hijos de *N* como subárboles hijos de *M*.

La siguiente proposición garantiza la corrección y completitud de la navegación basada en esta estrategia:

Proposición. *Sea T un árbol con raíz incorrecta. Entonces el método de navegación pregunta y divide finaliza en un número finito de pasos, señalando a un nodo crítico de T.*

Demostración.

Para garantizar la terminación basta con asegurar que el nodo *N* seleccionado en el caso |*T*| > 1 es siempre distinto de *R* = raíz (*T*), ya que en este caso el subárbol considerado al repetir el proceso tendría un número menor de nodos que el árbol original al tiempo que no sería el árbol vacío, lo que garantiza que en algún momento se alcanzará la condición de salida |*T*| = 1.

Para comprobar que *N ! = R* vamos a comprobar que para cualquier *M ∈ T, M != R*, se tiene |*dif* (*T, M*)| < |*dif* (*T, R*)|, de donde

|*dif* (*T, R*)| > *min{* |*dif* (*T, M*)| : *M ∈ T}*

y *R* no podrá ser el nodo *N* elegido. Sea entonces *M ∈ T*, *M !=R* (al menos un nodo de esta forma existe al estar considerando |*T*| > 1). Se cumple |*dif* (*T, M*)| < |*dif* (*T, R*)| porque

- |*dif* (*T, R*)| = |*T*| , ya que *subárbol* (*T, R*) = *T*.

- $|dif\,(T, M)| < |dif\,(T, R)|$ ya que al ser $M \mathrel{!}= R$ se cumplen
  - $0 < (|T| - |subárbol\,(T, M)|) < |T|$
  - $0 < |subárbol(T, M)| < |T|$

de donde

$$-|T| < (|T| - |subárbol\,(T, M)|) - |subárbol\,(T, M)| < |T|$$

es decir,

$$|dif\,(T, M)| < |T|.$$

Ahora debemos comprobar que el nodo señalado es efectivamente un nodo crítico. Para esto vamos a proceder por inducción sobre $|T|$. Si $|T| = 1$ al tener raíz no válida ésta es crítica y así lo indica la estrategia. Si $|T| > 1$ tenemos que distinguir dos casos:

1.- Si el nodo seleccionado *N* es no válido el algoritmo considera recursivamente *subárbol (T, N)* y por hipótesis de inducción se encontrará un nodo crítico en este subárbol, que también será entonces crítico en *T*.

2.- Si por el contrario *N* es válido se elimina su subárbol y se considera el árbol resultante *T' = T – subárbol (T, N)*. *T'* continúa teniendo la raíz no válida porque hemos probado que *N != R* y un número menor de nodos, por lo que la estrategia encontrará un nodo crítico *B* $\in$ *T'*. Ahora bien, debemos asegurar que este nodo *B* es también crítico en *T*. Pero para ello basta con observar que la única diferencia entre los hijos de *B* en *T'* y los hijos de *B* en *T* es que quizás en éste último caso haya un nodo más, la raíz *N* del subárbol eliminado, que al ser válida no altera la condición de nodo crítico de *B*.

REFERENCIAS

.
[1] G. Ferrand. *Error Diagnosis in Logic Programming, an Adaptation of E.Y. Shapiro's Method*. The Journal of Logic Programming 4(3):177{198, 1987.

[2] J.W. Lloyd. *Declarative Error Diagnosis*. New Generation Computing 5(2):133-154, 1987.

[3] L. Naish. *A Declarative Debugging Scheme*. Journal of Functional and Logic Programming, 1997-3.

[4] H. Nilsson. *How to look busy while being lazy as ever: The implementation of a lazy functional debugger*. Journal of Functional Programming 11(6):629-671.

[5] H. Nilsson y J. Sparud. *The Evaluation Dependence Tree as a basis for Lazy Functional Debugging*. Automated Software Engineering, 4(2):121-150, 1997.

[6] B. Pope y L. Naish, *Practical Aspects of Declarative Debugging in Haskell 98*, In Proc. PPDP03, ACM Press, 230-240, 2003.

[7] E.Y.Shapiro. *Algorithmic Program Debugging*. The MIT Press, Cambridge, Mass., 1982

[8] A. Tessier y G. Ferrand. *Declarative Diagnosis in the CLP Scheme*. En P. Deransart, M. Hermenegildo y J. MaÃluszynski (Eds.), Analysis and Visualization Tools for Constraint Programming, Chapter 5, 151-174. Springer LNCS 1870, 2000.

[9] Técnicas de diagnóstico y depuración declarativa para lenguajes lógico-funcionales . Rafael Caballero. Tesis Doctoral. Junio 2004.

[10] *jsqlparser.sourceforge.net. Libreria JSQLParser*